Advanced Data Structures and Algorithm Analysis

Laboratory Project 5

Diff

Zhao Bingqian, Lei Lulu, Lin Xiaojun

Date: 2011-1-2

Catalogue

Chapter 1: Introduction

1.1 General Problem: LCS

In biological applications, we often want to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called bases, which are represented by A, C, G and T. One goal of comparing two strands of DNA is to determine how "similar" the two strands are. One way is to find a third strand, whose bases appear in each of the first two strand and these bases must appear in the same order. [2]

Here comes the longest common subsequence(LCS) problem. It is a classic computer science problem, which is widely used in bioinformatics and mentioned in the first paragraph. And it is the basis of this project diff.

1.2 Problem Description

In computing, *diff* is a file comparison utility that outputs the differences between two files. It is typically used to show the changes between a file and a former version of the same file. Diff displays the changes made per line for text files.

For the given inputs, you should give the outputs as below:

Sample Input:

```
1 1
This is a test
This is a test
1 1
ab
cd
1 1
dynamic
programming
4 4
This is a test
which is more complicated
zzz
than ...
This is another test
of the project
which is much more complex
than the previous one
-100
```

Sample Output:

```
No difference found
Totally different
3
line #1:
dyn-progr+m+c-ng+
39
line #1:
nother+
line #2:
of the project+
much+icat-dline
#3:
zzzline
#4:
x+
...-the previous one+
```

1.3 What we do

For each test case, we first connect all the lines into one line of each file (including the newlines). Second, we use Dynamic Programming to get the LCS length of the two files, while an N*M table is used to store the length of LCS of each state. Third, according to the state table, we can get the path of state transition which is stored into an array. At last, we print out the answer in diff-like format.

The testing results are provided in Chapter 3.

What’s more, we analyse the algorithm and make further comments on the problem, which you can find in Chapter 4.

1.4 Algorithm Background: Dynamic Programming

The term dynamic programming was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. Now, dynamic programming has long been applied to numerous areas in mathematics, science, engineering, business, medicine, information systems, biomathematics, artificial intelligence, among others.

In mathematics and computer science, dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems which are only slightly smaller and optimal substructure. This method is based on Bellman's Principle of Optimality, which he phrased as follows.

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

The principle of optimality is also known as the "optimal substructure" property in the literature.[3][4]

Chapter 2: Data structure and Algorithm Specification

2.1 Data Structure

We use two string array to store the two files. An N*M table is used to store the states, the length of LCS. To save space cost, we use an N+M array to store the state transition instead of another N*M table.

The pseudo-code of the data structure is provided as follow:

```
/*String is the string array to store the whole charaters of
a file.*/
typedef char String[MaxLength+1];
/*PathArray is the array to store the state trasition path.*/
typedef int PathArray[MaxLength*2+2];
/*Matrix is the dynamic array to store the length of the
states.*/
typedef int **Matrix;
```

2.2 DP Algorithm for LCS

The LCS problem has an optimal substructure: the problem can be broken down into smaller, simple "subproblems", which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial. The LCS problem also has overlapping subproblems: the solution to a higher subproblem depends on the solutions to several of the lower subproblems.[5]

We have this theorem:

Let $Z = \{z_1, \ldots, z_k\}$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.

3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}.

So, we can get the following recursive formulation:[2]

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \textit{ and } x_i = y_j, \\ \max(c[i-1,j], c[i,j-1]) & \text{if } i,j > 0 \textit{ and } x_i \neq y_j. \end{cases}$$

The pseudo-code of DP for LCS is provided as follow:

```
LCS_Length(x, y, n, m, c)
   //Initialize the edges of the state table.
   for(i=0;i<=n;i++)
      c[i][0]=0;
   for(i=1;i<=m;i++)
      c[0][i]=0;
   //Get the state value according to the former states.
   for(i=1;i<=n;i++)
      for(j=1;j<=m;j++)
         if(x[i]==y[j])
            c[i][j]=c[i-1][j-1]+1;
         else
            c[i][j]=max(c[i-1][j], c[i][j-1]);
```

Chapter 3: Testing Results

3.1 Testing Cases and Purposes

We use 12 special cases to test the correctness of the algorithm, and 10 large file of the best case and the worst case to test the time of the algorithm. The detailed test cases are included in the package.

Special cases:

No.	Input	Expected Output	Purpose
1	0 0	No difference found	Test when n=m=0
2	0 1 Hello	Totally different	Test when n=0, m≠0
3	1 0 Hello	Totally different	Test when n≠0, n=0
4	1 1 Hello	Totally different	Test when a line is empty
5	1 1 This is a test This is a test	No difference found	Test when the files are the same.
6	1 1 ab cd	Totally different	Test when the files are totally different
7	1 1 dynamic programming	3 line #1: dyn-progr+m+c-ng+	A simple case to test the connectness.
8	4 4 aaaa aaa aa a a aa aaa aaaa	9 line #1: a- line #2: line #3: line #4: a+	A simple case to test the connectness.

No.	Input	Expected Output	Purpose
9	2 1 ab cd abcd	4 line #1: line #2:	A simple case to test the connectness.
10	4 3 aaaa bc ef aaaa aaaa xy aaaa	8 line #1: line #2: bc-xy+ line #3: ef- line #4:	A simple case to test the connectness.
11	3 4 aaaa bc aaaa aaaa xy up aaaa	8 line #1: line #2: bc-xy+ line #3: up+	A simple case to test the connectness.
12	4 4 This is a test which is more complicated zzz than ... This is another test of the project which is much more complex than the previous one	39 line #1: nother+ line #2: of the project+ much+icat-d- line #3: zzz- line #4: x+ ...-the previous one+	A complex case to test the connectness.

Large cases:

Type	Description
The best case	All the characters of the two files are the same.
The worst case	All the characters of each file are the same, but different from the other file.

3.2 The Table of Results

For the large-segments cases, we use M=100 to test, cause in real case M will not be too large.

Special cases:

No.	Actual Output	Status	Reason
1	No difference found	Corrected	
2	Totally different	Corrected	
3	Totally different	Corrected	
4	Totally different	Corrected	
5	No difference found	Corrected	
6	Totally different	Corrected	
7	3 line #1: dyn-progr+m+c-ng+	Corrected	
8	9 line #1: a- line #2: line #3: line #4: a+	Corrected	
9	4 line #1: line #2:	Corrected	
10	8 line #1: line #2: bc-xy+ line #3: ef- line #4:	Corrected	
11	8 line #1: line #2: bc-xy+ line #3: up+	Corrected	

No.	Actual Output	Status	Reason
12	39 line #1: nother+ line #2: of the project+ much+icat-d-x+ line #3: zzz- line #4: ...-the previous one+	Wrong	The character ‘x’ is outputted before the line number is outputted.

3.3 The Running Time

We test the running time of the best case and the worst case. The data scale N*M is from 800*800 to 4000*4000.

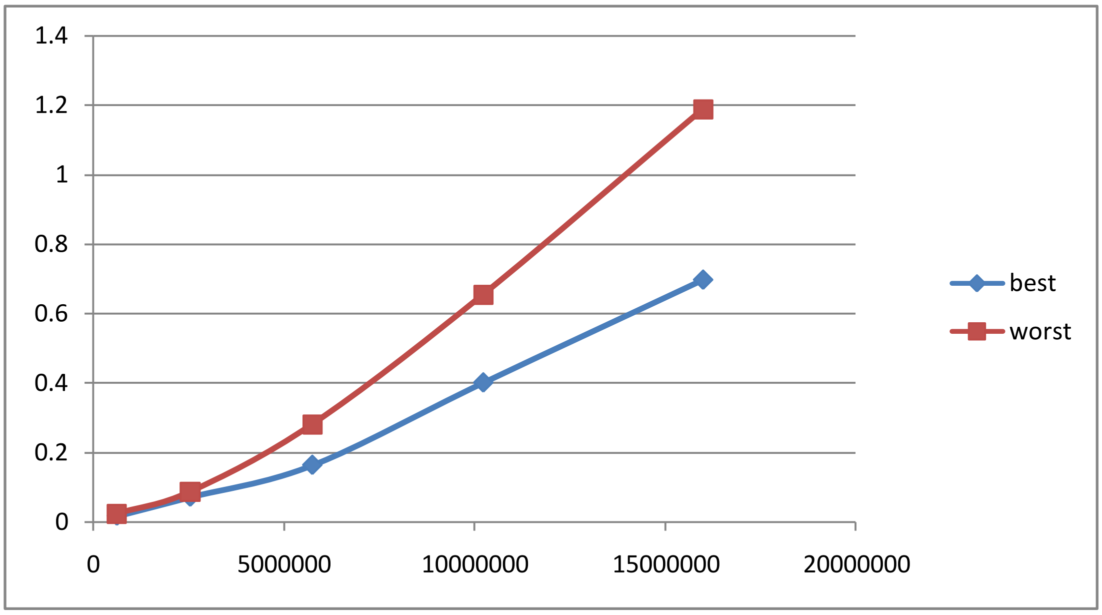

Figure 1 The running time of the best and worst cases

Chapter 4: Analysis and Comments

4.1 Analysis of Our Algorithm

The analysis of the time complexity is given as follows:

1. The DP algorithm check every state of the two files, so the time complexity is O(N*M).
2. The GetPath function follows the whole state transition path whose length is (N+M-CommonLength), so the time complexity is O(N+M).
3. The diff printing function follow the state transition path also, so the time complexity is the same to Step 2.

In conclusion, the time complexity of our algorithm is O(N*M).

The space complexity of our algorithm is O(N*M), because we use a N*M dynamic array to store the state value.

4.2 Comments on Our Algorithm

We use dynamic programming to get the LCS, which is the best algorithm as known. Than we get the path of the state transition, which make the diff printing much easier. Otherwise, printing the diff directly according to the state value table is too complexity to code, which we have tried but it doesn’t work well.

But there are still some bugs exist in our algorithm. We can’t get the same correct output of the last case in the sample input. One possible reason we guess is that our algorithm doesn't follow the project's idea exactly. However, we have tried our best and we have no more idea about the sample output of the project. If you have a better algorithm, you can connect us by e-mail to elf@zju.edu.cn.

There are some possible improvements of our algorithm.

1. As the DP just use the adjacent state, we may save the space cost by just using two one-dimensional array to store the state value, and using other data structures to store the state transition.
2. The output of the algorithm can also be improved. It can distinguish an English word and make sure as many as possible words will be matched.

Appendix: Source Code (in C++)

Source code of RichestPeople.cpp

```
/*
* cy_G5_P5.c - the program for Research Project 5 of Advanced Data Structures
and Algorithm Analysis.
* Copyright (C) Zhao Bingqian of Group 5. All rights reserved.
* Purpose: Print out the differences of two files in diff-like format.
* Algorithm: Using dynamic programming to get the LCS of the two files.
* Notice: The line number of each file should no larger than 50,
*       and the characters in a line should no more than 80.
* Instructor: Chen Yue.
* Version: 2011/1/3
*/
#include<stdio.h>
#include<string.h>

#define MaxLength 4500     /*The largest number of characters in a file.*/
#define LineLength 85      /*The largest number of characters in a line.*/

/*String is the string array to store the whole charaters of a file.*/
typedef char String[MaxLength+1];
/*Line is the string array to store a line*/
typedef char Line[LineLength+1];
/*PathArray is the array to store the state trasition path.*/
typedef int PathArray[MaxLength*2+2];
/*Matrix is the dynamic array to store the length of the states.*/
typedef int **Matrix;

/*The prior declaration of the functions.*/
void LCS_Length(String x, String y, int lx, int ly, Matrix c);
void GetPath(String x, String y, Matrix c, int i, int j, int k, PathArray path,
int *NextLine);
void Print_Diff(String x, String y, PathArray path, int PathLen);
void PrintChar(char c);
void PrintLine(int k);

int main(){
    /*string x,y store the characters of the two files.*/
    String x, y;
    /*TempStr is used to get a whole line of a file while inputting.*/
    Line TempStr;
    /*c is the dynamic array that store the state value.*/
    Matrix c;
    /*path is the array that store the state transition path.*/
    PathArray path;
    /*
    *ComNum is the whole matched characters number, including the newline.
    *NextLine is the matched newline number.
    *PathLen is the length of the state transition path.
    */
    int ComNum, NextLine, PathLen;
    /*
    *lx,ly are the length of the two file.
    *n,m are the line number of the two file.
    */
    int lx, ly, n, m;
    int i, j, k;

    /*New a space for dynamic array c.*/
    c=(Matrix)malloc((MaxLength+1)*sizeof(int*));
    for(i=0;i<=MaxLength;i++)
        c[i]=(int*)malloc((MaxLength+1)*sizeof(int));

    /*Read in the first file line number n first.*/
    scanf("%d",&n);
    /*If n is negative, stop read in new cases.*/
    while(n>=0){
        /*Read in the second file line number m.*/
        scanf("%d",&m);
        getchar();

        /*Handling the special cases.*/
        if(!m&&!n){
            puts("No difference found");
            scanf("%d",&n);
            continue;
        }
        if(!m||!n){
            puts("Totally different");
            m+=n;
            for(i=0;i<m;i++)
                fgets(TempStr,LineLength-1,stdin);
            scanf("%d",&n);
            continue;
        }
        /*n and m is normal case, give the normal answers.*/
        /*Read the first file line by line and copy them to a string start from
x[1].*/
        strcpy(x,"#");
        for(i=0;i<n;i++){
            fgets(TempStr,LineLength-1,stdin);
            strcat(x,TempStr);
        }
        lx=strlen(x)-1;

        /*Read the second file line by line and copy them to a string start from
x[1].*/
        strcpy(y,"#");
        for(i=0;i<m;i++){
            fgets(TempStr,LineLength-1,stdin);
            strcat(y,TempStr);
        }
        ly=strlen(y)-1;

        /*Calculate the length of the LCS.*/
        LCS_Length(x,y,lx,ly,c);
        /*Get the whole common characters number.*/
        ComNum=c[lx][ly];
        /*Calculate the length of state transition path.*/
        PathLen=lx+ly-ComNum;

        /*Record the state transition path, and count the common newline number.*/
        NextLine=0;
        path[0]=0;
        path[PathLen+1]=1;
        x[0]=y[0]=0;
        GetPath(x,y,c,lx,ly,PathLen,path,&NextLine);

        /*Print out the answer as required.*/
        if(!(ComNum-NextLine))
            puts("Totally different");
        else if(ComNum==lx&&ComNum==ly)
            puts("No difference found");
        else{
            printf("%d\n",ComNum-NextLine);
            Print_Diff(x,y,path,PathLen);
        }
        /*Read in the next n.*/
        scanf("%d",&n);
    }

    /*Free the space of dynamic array c.*/
    for(i=0;i<=MaxLength;i++)
        free(c[i]);
    free(c);
}

/*
*Dynamic programming to calculate the length of LCS.
*/
void LCS_Length(String x, String y, int lx, int ly, Matrix c){
    int i, j;

    /*Initialize the edge state value.*/
    for(i=0;i<=lx;i++)
        c[i][0]=0;
    for(i=1;i<=ly;i++)
        c[0][i]=0;

    /*Dynamic programming the state value from small to large.*/
    for(j=1;j<=ly;j++)
        for(i=1;i<=lx;i++){
            if(x[i]==y[j]){
                c[i][j]=c[i-1][j-1]+1;
            }
            else
                if(c[i-1][j]>c[i][j-1]){
                    c[i][j]=c[i-1][j];
                }
                else{
                    c[i][j]=c[i][j-1];
                }
        }
}

/*
*Get the state trasition path, and count the common newline number.
*path[k]=0 means in the kth step, it goes upleft;
*path[k]=-1 means in the kth step, it goes legt;
*path[k]=1 means in the kth step, it goes up.
*/
```

```c
void GetPath(String x, String y, Matrix c, int i, int j, int k, PathArray path,
int *NextLine){
    /*Stop until reached c[0][0].*/
    while(i||j){
        /*Consider a special situlation.*/
        if(!path[k+1]&&x[i]==y[j]&&x[i+1]!='\n'&&x[i]!='\n'){
            /*Mark the path value as 0.*/
            path[k]=0;
            i--;
            j--;
            k--;
        }
        /*If it can go left, go left first.*/
        else if(j&&c[i][j]==c[i][j-1]){
            /*Mark the path value as -1.*/
            path[k]=-1;
            j--;
            k--;
        }
        /*Otherwise, if it can go up, then go up.*/
        else if(i&&c[i][j]==c[i-1][j]){
            /*Mark the path value as 1.*/
            path[k]=1;
            i--;
            k--;
        }
        /*Otherwise, it can go upleft only.*/
        else{
            /*Mark the path value as 0.*/
            path[k]=0;
            /*If the common character is a newline, count it.*/
            if(x[i]=='\n')(*NextLine)++;
            i--;
            j--;
            k--;
        }
    }
}

/*
*Print out the changes from the first file to the second.
*/
void Print_Diff(String x, String y, PathArray path, int PathLen){
    /*
    *Variable f mark the former print state.
    *Variable fl mark print state of a new line.
    *Variable line count the present line number.
    */
    int i,j,k,f,fl,line;

    /*Print out the first line number.*/
    line=1;
    PrintLine(line);
    /*Initialize the flags.*/
    f=fl=0;
    /*
    *Print out one by one according to the path.
    *Start from x[1], y[1].
    */
    i=j=0;
    for(k=1;k<=PathLen;k++){
        /*If they are common characters.*/
        if(!path[k]){
            i++;
            j++;
            /*If the former print a character to be deleted, ptint out a '-'.*/
            if(f>0)
                PrintChar('-');
            /*If the former print a character to be added, ptint out a '+'.*/
            else if(f<0)
                PrintChar('+');
            /*If the common character is a newline.*/
            if(x[i]=='\n'){
                /*If something printed in this line, turn to a new line.*/
                if(fl)
                    PrintChar('\n');
                /*Print out the new line number.*/
                if(x[i+1]){
                    line++;
                    PrintLine(line);
                }
                /*Refresh the line flag.*/
                fl=0;
            }
            /*Refresh the print flag.*/
            f=0;
        }
        /*Otherwise, if a character should be deleted.*/
        else if(path[k]>0){
            /*If the former print is an added character, print out a '+'.*/
            if(f<0){
                PrintChar('+');
                /*Refresh the print flag.*/
                f=0;
            }
            i++;
            /*If the character to be added isn't a newline, print it out.*/
            if(x[i]!='\n'){
                PrintChar(x[i]);
                /*Mark the print flag and line flag.*/
                f=fl=1;
            }
            /*Otherwise, it's a newline.*/
            else{
                /*
                *If the former printed is deleted character,
                *print out a '-' and turn to a new line.
                */
                if(f){
                    PrintChar('-');
                    PrintChar('\n');
                    /*Refresh the print flag.*/
                    f=0;
                }
                /*Print out the new line number.*/
                if(x[i+1]){
                    line++;
                    PrintLine(line);
                }
                /*Refresh the line flag.*/
                fl=0;
            }
        }
        /*Otherwise, the character should be added.*/
        else{
            /*If the former print is a deleted character, print out a '-'.*/
            if(f>0){
                PrintChar('-');
                /*Refresh the print flag.*/
                f=0;
            }
            j++;
            /*If the character to added isn't a newline, print it out.*/
            if(y[j]!='\n'){
                PrintChar(y[j]);
                /*Mark the print flag and line flag.*/
                f=-1;
                fl=1;
            }
            /*Otherwise, it's a newline.*/
            else
                /*
                *If the former printed is added character,
                *print out a '+' and turn to a new line.
                */
                if(f){
                    PrintChar('+');
                    PrintChar('\n');
                    /*Refresh the print flag and line flag.*/
                    f=fl=0;
                }
        }
    }
}

/*Print out a single char.*/
void PrintChar(char c){
    putchar(c);
}

/*Print out the line number.*/
void PrintLine(int k){
    printf("line #%d:\n",k);
}
```

References

[1] Mark Allen Weiss, “Data Structure and Algorithm Analysis in C (Second Edition)”, *POSTS&TELECOM PRESS,* 2005

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, “Introduction to Algorithms (Second Edition)”, *MIT Press and McGraw-Hill Higher Education,* 2001

[3] Art Lew and Holger Mauch, “Dynamic Programming: A Computational Tool”, *Springer-Verlag Berlin Heidelberg,* 2007

[4] Wikipedia, “Dynamic programming”, http://en.wikipedia.org/wiki/Dynamic_programming

[5] Wikipedia, “Longest common subsequence problem”, http://en.wikipedia.org/wiki/Longest_common_subsequence_problem

[6] Wikipedia, “Diff” , http://en.wikipedia.org/wiki/Diff

Author List

Zhao Bingqian, the program coder;

Lei Lulu, the project tester;

Lin Xiaojun, the report writer.

Declaration

We hereby declare that all the work done in this project titled " Diff" is of our independent effort as a group.

Signatures

赵冰骞，雷露露，林小均