FileDistance

Relazione Sistemi Operativi Laboratorio Anno 2019-2020

Sommario

Descrizione del Progetto

Il progetto FileDistance è un'applicazione scritta in c che consente all'utente di calcolare la distanza di edit a uno o più file.

La distanza di edit è una funzione che consente di verificare quanto due stringhe (o sequenze di byte) siano lontane una dall’altra. Questa distanza viene calcolata sulla base del numero di operazioni necessarie a trasformare una stringa nell’altra.

Le operazioni sono:

- Aggiungere un carattere/byte;
- Eliminare un carattere/byte;
- Modificare un carattere/byte.

Funzionalità

Le funzionalità sono 5;

- ./filedistance distance file1 file2

Il programma stampa la distanza di edit tra i due file e il tempo impiegato per calcolarla.

- ./filedistance distance file1 file2 output

Il programma stampa la distanza di edit tra i due file, il tempo impegato per calcolarla e scrive sul file output le istruzioni in binario.

- ./filedistance apply file1 filem output

Il programma applica al file1 le modifiche scritte nel filem e le sovrascrive al file output

- ./filedistance search file1 dir

Restituisce in output i file contenuti in dir (e nelle sue sottodirectory) che hanno minima distanza da inputfile .Il path assoluto di ogni file viene presentato in una riga dello standard output.

- ./filedistance searchAll inputfile dir limit

Vengono restituiti in output tutti i file contenuti in dir (e nelle sue sottodirectory) che hanno una distanza da inputfile minore o uguale di limit. I file vengono presentati nello standard output in ordine crescente secondo il seguente formato:

distanza pathassolutofile

Librerie

file_distance.h

La libreria file_distance.h è il cuore del progetto, la libreria centrale che utilizzando le varie librerie fornisce le informazioni richieste. Al suo interno sono presenti 5 metodi che vanno a richiamare funzioni in file_modifier.h, leven.h e finder.h

file_handler.h

La libreria file_handler.h ha come scopo principale quelli di eseguire operazioni su file, i due metodi principali sono changesApply che va a leggere da file le istruzioni dal filem e le applica al file in ingresso, e getStringFromFile che, dato un file, prende il suo contenuto e lo inserisce in una stringa.

finder.h

La libreria finder.h si occupa delle operazioni tra le varie cartelle, e applica levensthein ai vari file, inoltre contiene anche la lista concatenata che contiene le informazioni ricavate dai vari file e cartelle.

istruction_structure.h

La libreria istruction_structure.h ha al suo interno due strutture dati: MaxHeap e IstructionData, le due strutture vengono utilizzate per memorizzare le istruzioni necessarie per il completamento delle operazioni. Le varie funzioni servono ad aggiungere e gestire questi elementi. Inoltre contiene anche l'ENUM utilizzato per definire il tipo di operazione da svolgere. Probabilmente contiene l'errore di segmentazione che rende quasi inutilizzabile il programma.

leven.h

La libreria leven.h contiene l'algoritmo di Levensthein che permette di calcolare la distanza di edit. Le due funzioni presenti consentono di trovare la distanza di edit e oltre ad essa anche di scrivere le istruzioni da utilizzare su file.

timer.h

La libreria timer.h contiene 3 metodi che consentono di misurare i secondi impiegati per svolgere l'algoritmo.

Difficoltà riscontrate e conseguenze

Nel corso dello sviluppo dell'applicazione sono state riscontrate molte problematiche, inizialmente, lo sviluppo è stato svolto senza l'utilizzo di un debugger per via dell'impossibilità di installare cmake, ciò ha causato un forte rallentamento nello sviluppo. Successivamente, una volta implementato sia levensthein che la scrittura e la lettura da file completamente funzionante, è venuto alla luce un bug nell'edit distance delle stringhe con più lettere uguali e per risolvere il problema, ho riscritto completamente leven.c, una volta terminato il programma non era più utilizzabile e non sono riuscito a localizzare la causa dell'errore di segmentazione nato in quel momento. Per risolvere il problema ho deciso di cambiare architettura ed utilizzare un debugger, ma cambiando pc l'errore di segmentazione non era segnalato, e il programma era funzionante, ma il tempo investito dietro a questa operazione è stato molto e ha comportato un forte impatto sulla qualità delle operazioni svolte successivamente. A causa di tutto ciò il codice è funzionante solamente su sistema operativo MacOS, me non in uno linux.

Strutture Dati Utilizzate

MaxHeap

Per memorizzare le istruzioni ho implementato un MaxHeap poichè le istruzioni al nuovo file andranno applicate in ordine decrescente, e questa struttura dati consente di estrarre il valore massimo contenuto in essa in un tempo costante **Θ**(1), un costo di inserimento di O(log2n) ed un costo di ordinamento di MaxHeapify di O(n).

```c
/**
 * Sotto-Struttura contenente le informazioni delle istruzioni che saranno
utilizzate per apportare modifiche al file.
 */
struct istructionData
{
    /**
     * Tipo di istruzione da apportare: ADD, SET o DEL
     */
    IstructionType istruction;
    /**
     * Unsigned int che indica la posizione in cui verrà apportata la modifica
     */
    int position;
    /**
     * Parametro utilizzato solamente nei metodi ADD e SET, che indica il nuovo
valore che avrà la lettera nella posizione position
     */
    char letter;
};
typedef struct istructionData IstructionData;

/**
 * MaxHeap contenente le istruzioni per la modifica del file.
 */
struct MaxHeap
{
     /**
     * Grandezza del MaxHeap
     */
    int size;
    /**
     * Numero di elementi presenti al suo interno
     */
    int count;
    /**
     * Istruzione contenuta
     */
    IstructionData *data;
};
typedef struct MaxHeap MaxHeap;
```

ScanData

Per quanto riguarda la decisione di utilizzare una lista concatenata per memorizzare i dati presi dallo scan ricorsivo, è basata principalmente sul tempo di implementazione, e non

sull'ottimizzazione. Una decisione migliore sarebbe stata quella di implementare un MinHeap poichè le distanze vanno restituite in ordine crescente, ma date le tempistiche impiegate per la prima struttura, ho preferito evitare. Lo stesso discorso vale per l'algoritmo di ordinamento, ovvero il bubblesort.

```
/**
 * Struct contenente path, distanza e puntatore al nodo successivo della lista.
 */
struct ScanData{
    /**
     * Path del file
     */
    char *path;
    /**
     * distanza del file dal file in input
     */
    int distance;
    /**
     * Punatore all'elemento successivo
     */
    struct ScanData *next;
};
typedef struct ScanData ScanData;
```

Levensthein

Nella mia implementazione dell'algoritmo di Levenshtein ho utiizzato una due metodi simili che producono una matrice dinamica e la compilano seguendo i parametri dell'algoritmo di Levenshtein inserendo nella casella della matrice il minimo tra i valori presenti nelle tre celle adiacenti alto - diagonale sinistra - sinistra.

Per riempire la matrice ho utilizzato il metodo fillMatrix:

```
int ** fillMatrix(int **matrix, char *toModify, int nrighe, char *finalResault,
int ncolonne)
{
   int costo = 0;
    for ( int i = 1; i < nrighe; i++)
    {
        for ( int j = 1; j < ncolonne; j++)
        {
            if (toModify[i - 1] == finalResault[j - 1])
                costo = 0;
            else
                costo = 1;
            matrix[i][j] = min(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1,
matrix[i - 1][j - 1] + costo);

        }
    }
    return matrix;
}
```

La differenza tra i due metodi sta nel "tragitto" che levensthein_distance_out va a tracciare per ricostruire le istruzioni di edit che verranno poi applicate al primo file per essere poi trasformato nel secono.

Per la ricerca delle istruzioni ho utilizzato un metodo ricorsivo recFind.

```
void recFind(int ** matrix, MaxHeap * hp, int riga, int colonna, char *
toModify, char * finalResault)
{
    if (matrix[riga][colonna] != 0)
    {
        if(riga -1>=  0 && colonna-1>= 0)
        {
        int app = min(matrix[riga][colonna-1],matrix[riga-1][colonna-
1],matrix[riga-1][colonna]);
        if(matrix[riga -1 ][colonna - 1] == app)
        {
            // ! SET
            if(matrix[riga][colonna] != matrix[riga-1][colonna-1])
            {
            pushIstruction(hp, SET, colonna-1, finalResault[riga-1]);
                printf("\nSET%d%c",colonna - 1,finalResault[riga-1]);
            }
            recFind(matrix, hp, riga - 1, colonna -1,toModify, finalResault);
        }
        else if (matrix[riga][colonna -1] == app && colonna-1 >= 0)
        {
            // ! ADD
            pushIstruction(hp, ADD, riga-1, finalResault[riga-1]);
            printf("\nADD%d%c",riga-1,finalResault[riga-1]);
            recFind(matrix, hp, riga, colonna-1, toModify, finalResault);
        }
        else if(matrix[riga-1][colonna] == app && riga-1 >= 0)
        {
            // ! DEL
            pushIstruction(hp, DEL, colonna,'-');
            printf("\nDEL%d", colonna);
            recFind(matrix, hp, riga-1, colonna, toModify, finalResault);
        }


        }
        else if (riga - 1 < 0)
        {
            // ! ADD
            pushIstruction(hp, ADD, riga-1, finalResault[riga-1]);
            printf("\nADD%d%c",riga-1,finalResault[riga-1]);
            recFind(matrix, hp, riga, colonna-1, toModify, finalResault);
        }
        else if (colonna - 1 < 0)
        {
            // ! DEL
            pushIstruction(hp, DEL, colonna,'-');
```

```
            printf("\nDEL%d", colonna);
            recFind(matrix, hp, riga - 1, colonna, toModify, finalResault);
        }
        else return;
    }
    else return;
}
```

Per ottimizzare il costo computazionale dell'algoritmo di Levensthein ho aggiunto il metodo complexityReduction che si occupa di cancellare le ultime lettere di ogni stringa in caso risultino uguali, sviluppando così una matrice più piccola, con conseguente riduzione dell'uso memoria e di processore.

```
void complexityReduction(char *stringa1, char *stringa2)
{
    if(stringa1[strlen(stringa1)-1] == stringa2[strlen(stringa2)-1])
    {
        stringa1[strlen(stringa1)-1] = 0;
        stringa2[strlen(stringa2)-1] = 0;
        complexityReduction(stringa1, stringa2);
    }
    else return;
}
```

Modularità

Il programma è basato su 3 moduli fondamentali utilizzati dalla libreria "file_distance.h".

Il primo modulo è "leven.h", una libreri a che consente di applicare l'algoritmo di Leven in due modi: il primo genera solamente una matrice e la va a riempire in modo tale da ottenere la distanza di edit delle due stringhe, mentre il secondo consente anche di avere la sequenza di istruzioni necessarie per trasformare il primo file nel secondo.

Il secondo modulo è "file_modifier.h", una libreria che raggruppa tutte le operazioni che si andranno ad eseguire nei file.

Il terzo modulo è "finder.h", questa libreria si occupa di gestire l'applicazione dei vari metodi di leven sulla directory corrente e le sue sottodirectory, salvando le informazioni ricevute dai vari leven in una lista concatenata.