Universidade do Porto
FEUP Faculdade de Engenharia

Conceção e análise de algoritmos

Plagiarism Checker

Fábio Filipe Jesus Silva - ei11107

Fernando Manuel Rocha Magalhães - ei07046

José Pedro Lobo Marinho Trocado Moreira - ei12002

Porto, 28 de maio de 2013

Resumo

Este trabalho contextualiza-se num sistema de deteção de plágio de ficheiros de texto.
Tal deteção é feita através do conceito de **subsequência comum mais comprida**, que recorre a programação dinâmica seguido de um algoritmo do tipo do "diff" frequentemente utilizado nos sistemas unix. Este segundo algoritmo pode ser implementado de forma recursiva ou não. Optamos por uma implementação não recursiva após termos verificado que a versão recursiva facilmente alcançava a profundidade máxima de recursividade para ficheiros de texto grandes. A versão não recursiva mostrou ainda ser ligeiramente mais rapida.

Este sistema compara um ficheiro de texto com uma base de dados (neste caso um diretório com vários ficheiros de texto) e calcula o grau de similaridade entre ambos.

Sendo assim, os dados de entrada são, portanto, os caminhos para a base de dados e para o ficheiro de texto. Tais caminhos podem ser inseridos das seguintes maneiras:

- Manualmente (pedido pelo programa em tempo de execução)
- "Arrastando" a pasta e o ficheiro para o executável (*Windows*)
- Como parâmetros da linha de comandos (*Windows* e *Linux*)

Por uma necessidade de desenvolvimento o programa foi desenvolvido usando compilação condicional por forma a fazer uso da WIN32 api (em Windows) ou da api de standard POSIX disponibilizada pelos sistemas Linux, nomeadamente do que diz respeito a acesso a ficheiros/directorias e verificações relacionadas com os mesmos

Em termos de resultado final, espera-se que o programa devolva (na stdout), para cada um dos ficheiros na base de dados:

- *Strings* similares às do ficheiro a testar editatas de forma a tornar perceptivel as partes similares entre o ficheiro testado e o ficheiro da base de dados
- Algumas estatisticas relativas ao processamento do ficheiro, nomeadamente número de caracteres, tempo de execução, grau de similaridade entre ficheiros, entre outras estatisticas de menor relevância.

Universidade do Porto
FEUP Faculdade de Engenharia

Índice

Principais algoritmos

O programa durante a sua execussão compara cada ficheiro da base de dados com o ficheiro a testar e produz um output com algumas estatisticas bem como os conteudos do ficheiro a testar editatos por forma a tornar visiveis as semelhanças entre os dois ficheiros testatos (o ficheiro a testar e o ficheiro da base de dados). O processo de comparação dos dois ficheiros consiste em duas fases com algoritmos que seguidamente descrevemos:

1. **Algoritmo LCS (Longest Common Subsequence) com recurso a programação dinâmica:**

   Este algoritmo, vastamente conhecido usa uma matriz (n+1) x (m+1) onde m é a dimensão (em número de caracteres) de um ficheiro e n a dimensão do outro. A primeira linha e coluna são preenchidas com zeros. Seguidamente da esquerda para a direita e de cima para baixo as células preenchem-se sequenciamente com o critério abaixo descrito (imaginando que estamos na célula da linha x, coluna y)

   1. Se a letra na posicão y-1 no ficheiro 1 é igual a letra x-1 do ficheiro 2 então a celula ficará com o valor da célula na linha x-1 coluna y-1 acrescentando-lhe 1. Passamos para a célula seguinte.
   2. Caso a condição anterior não se verifique a célula é preenchida com o maior valor de entre a célula imediatamente a sua esquerda e a célula imediatamente em cima de si. Passamos então para a célula seguinte.

   A **complexidade temporal** do algoritmo é facil de verificar que é **O(n*m)** uma vez que a operação mais repetida é a de preenchimento das células e esta é efectuada aproximadamente n*m vezes.

   A **complexidade espacial** é também de n*m pois corresponde á utilização da matriz acima referida

2. **Algoritmo do tipo “diff” (simplificado) para extração de informação da matriz gerada pelo algoritmo anterior:**

   Este algoritmo, menos divulgado do que o LCS acima referido percorre a matriz desde a ultima célula até á primeira de forma a extrair a informação contida nela.

   Utiliza como estruturas adicionais uma stack que no maximo atingirá o tamanho corresponnte ao numero de caracteres do ficheiro a testar (ficheiro 1). Utiliza ainda uma string para guardar alguns caracteres intermedios, esta string terá no maximo tamanho h onde h é o numero de caracteres comuns entre o ficheiro 1 e o ficheiro 2

   O algoritmo que percorre a matriz do direita para a esquerda de cima para baixo funciona da seguinte maneira estando nos na célula da linha x, coluna y:

   1. Se a letra na posicão y-1 no ficheiro 1 é igual a letra x-1 do ficheiro 2 então colocamos a letra na string auxiliar e na proxima iteração vamos para a célula da linha x-1 coluna y-1.
   2. Se a condição assima não se verifica e estamos já com y<0 esvaziamos a string auxiliar (este processo será descrito em baixo em mais detalhe). Passamos todos os caracteres que ainda não foram passados do ficheiro um para a stack (do fim do ficheiro para o inicio). Terminamos a execução do algoritmo
   3. Se nenhuma das condições acima ocorrer e se a celula á nossa esquerda tiver um valor maior do que a em cima de nós esvaziamos a string temporária. Colocamos o caracter na posição y-1 do ficheiro 1 na stack e passamos para a célula linha x coluna y-i na proxima iteração.

Se nenhuma das condicoes acima ocorrer e se a celula em cima da actual for maior ou igual á celula imediatamente á esquerda, esvaziamos a string temporária. Na proxima iteração iremos para a célula na linha x-1 coluna y.

Este processo é executado até que todos os caracteres do ficheiro 1 sejam colocados na stack do fim para o inicio.

O **processo de esvaziamento da string temporária** consiste em verificar se ela contem elementos suficientes para ser considerada uma secção que é semelhante entre ficheiros, (este número mínimo de caracteres pode ser especificado pelo utilizador antes do algoritmo executar através de um menu criado para o efeito). Caso tenha tamanho suficiente coloca-se o conteudo na string acompanhado de uma marcação para informar o utilizador de que se trata de uma secção encontrada nos dois ficheiros. Caso não tenha tamanho suficiente passam-se somente todos os elementos da string para a stack. Esta parte do algoritmo vai também tomando conta de quantos caracteres marca como pertencentes a ambos os ficheiros e retorna o valor no fim para ser mostrado ao utilizador.

Em termos de **complexidade temporal** é facil de verificar que este algoritmo percorre a matriz começãndo pelo fim da mesma até chegar na pior das hipoteses ao primeiro elemento da matriz. Isto é feito recorrendo a passos de uma casa ou na diagonal, ou de um passo para cima ou de um passo para a esquerda, sendo que no pior dos casos o algoritmo percorrerá a matriz seguindo para a casa a cima até não poder avançar mais e depois percorrerá a matriz para a esquerda até chegar á casa (0,0), ora isto levará a uma complexidade **O(m+n)**. Esta complexidade está no entanto muito dependente do numero de “matches” entre os dois ficheiros sendo que no limite (quando o ficheiro 1 é igual ao ficheiro 2) é de **O(n)**. Podemos ainda dizer que para dois ficheiros de tamanho semelhante n+m é aproximadamente 2n e portanto temos uma complexidade temporal de **O(n)**.

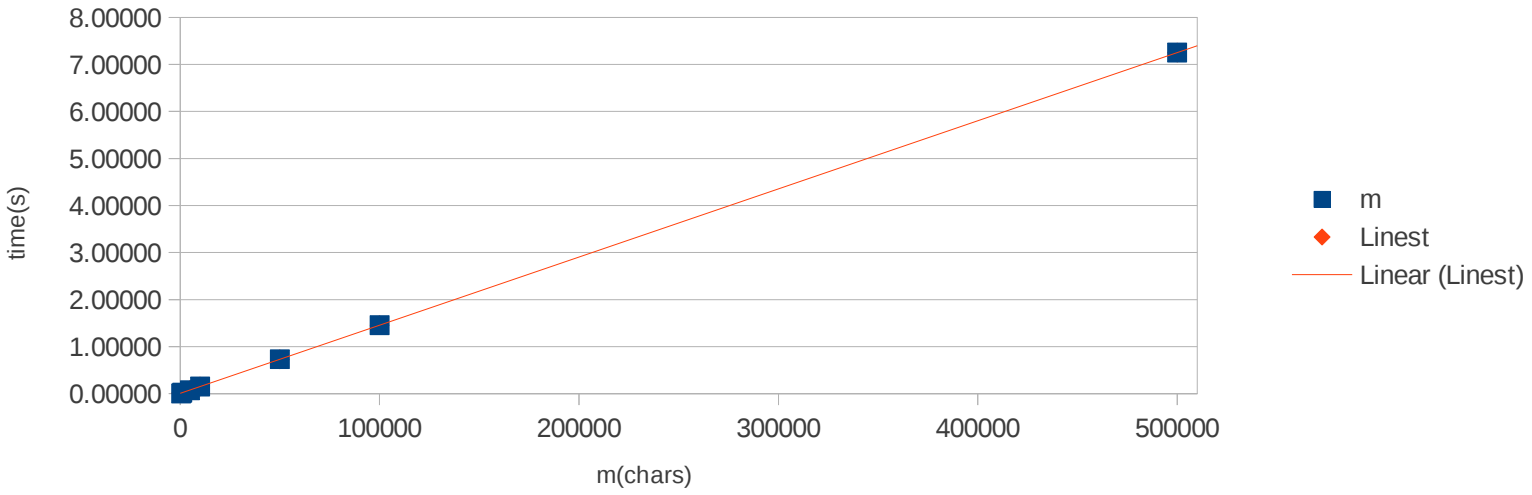
Quanto a **complexidade espacial** temos como estruturas utilizadas a stack que terá um **tamanho maximo de n** e a string auxiliar que terá tamanho h como acima enunciado, pelo que a complexidade espacial será de aproximadamente **n** pois a string auxiliar terá no maximo n elementos (aproximadamente).

Estes dois algoritmos são executados em sequência pelo que a **complexidade temporal do conjunto** não é mais do que a complexidade temporal do mais relevante de entre eles. Como para o primeiro algoritmo temos **O(m*n)** e para o segundo temos **O(m+n)** podemos afirmar que a complexidade temporal do conjunto é de aproximamente **O(m*n)**.

Em termos de c**omplexidade espacial do conjunto** temos como estruturas utilizadas a matrix de **n*m** elementos e a stack de apenas **n** elementos pelo que a complexidade do conjunto será de **n*m.**

No que há complexidade temporal diz respeito usando as estatisticas fornecidas pelo programa desenvolvido podemos obter os seguintes dados variando m e mantendo n constante:

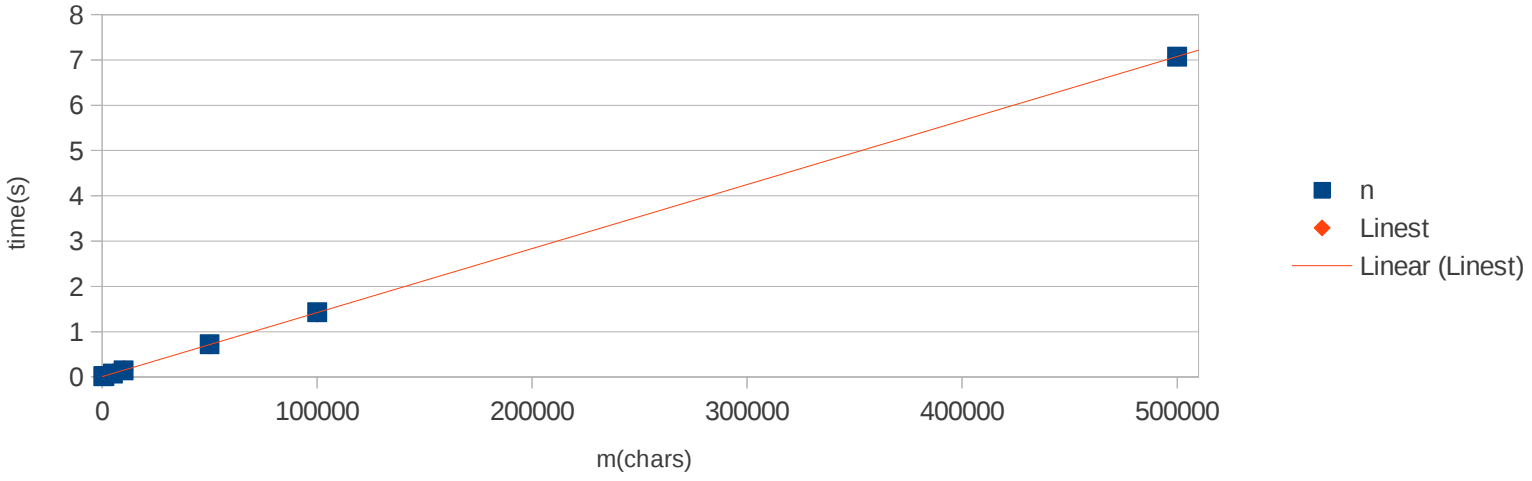
n	m	t1	t2	t3	t medio	Linest
1000	500	0.00612	0.00644	0.00611	0.00622	0.01133
1000	1000	0.02318	0.02088	0.02002	0.02136	0.01857
1000	5000	0.07416	0.07308	0.07214	0.07312	0.07656
1000	10000	0.14692	0.15066	0.14649	0.14802	0.14904
1000	50000	0.74599	0.72579	0.72760	0.73312	0.72889
1000	100000	1.45268	1.47853	1.44085	1.45735	1.45371
1000	500000	7.21064	7.30310	7.23968	7.25114	7.25224



Como os dados demonstam a **dependencia temporal** em m é de facto **linear** tal como afirmado anteriormente. É possivel ver o enquadramento quase perfeito dos dados obtidos no **ajuste linear** feito.

Resultados semelhantes foram obtidos quando variamos **n** (tamanho do ficheiro a ser testado) e mantivemos **m** constante, como a tabela e grafico a baixo mostram:

n	m	t1	t2	t3	t medio	Linest
500	1000	0.0124991	0.018688	0.0132699	0.01482	0.01633
1000	1000	0.0190439	0.0234611	0.018604	0.02037	0.02340
5000	1000	0.0719039	0.0767372	0.075954	0.07487	0.07994
10000	1000	0.148679	0.148971	0.141467	0.14637	0.15061
50000	1000	0.722559	0.725057	0.725392	0.72434	0.71598
100000	1000	1.43711	1.42594	1.42826	1.43044	1.42270
500000	1000	7.0772	7.10339	7.04201	7.07420	7.07644



É mais uma vez notória a **dependencia linear**, desta vez em **n** materializada na consistência existente entre os dados obtidos e o **ajuste linear**.

Em suma, com os dados obtidos pudemos **confirmar** a análise teorica que anteriormente tinhamos feito.

Principais dificuldades e esforço individual

Em geral, este segundo projeto foi mais facil do que o primeiro, com menos trabalho para fazer e melhor organização e articulação do grupo.

O mais trabalhoso e dificil no que á realização do trabalho diz respeito foi a arquitectura do programa para que pudesse funcionar quer sobre a api do Windows que sobre a api Posix (Linux), este problema foi solucionado com o uso de directivas ao pre-processador para uso de compilação condicional.

Foi também particularmente dificil fazer o estudo da complexidade do algoritmo do tipo “diff” pois é um algoritmo um pouco trabalhoso de analizar.

Quanto ao trabalho desempenhado por cada membro do grupo, esse foi equitativo, com distribuição equitativa de tarefas.

Conclusão

Em suma, podemos afirmar que este sistema de deteção de plágio comporta-se de maneira bastante satisfatória, pois apresenta taxas de similaridade bastante realistas. Sabemos no entanto que para grandes bases de dados e documentos de tamanho muito grande este algoritmo apresentaria resultados não tão satisfatório, motivo pelo qual tipicamente os sistemas deste tipo utilização outros métodos baseados em “fingerprints” e outros metodos de amostragem que apresentam resultados mais satisfatórios quando lidamos com grandes quantidades de dados.

Foi um trabalho em que todos os elementos do grupo concordam que aprenderam bastante sobre os assuntos em causas e onde pensamos que atingimos todos os objectivos pretendidos.

Bibliografia

http://en.wikipedia.org/wiki/Diff - Página *wiki* sobre o comando ***"diff"*** (*GNU/Linux*), que se assemelha ao nosso projeto

http://en.wikipedia.org/wiki/Longest_common_subsequence_problem - Página *wiki* sobre o método LCS (*Longest Common Sequence*), no qual se baseia o parte do nosso projecto.

http://www.xmailserver.org/diff2.pdf – PDF com análise de complexidade do algoritmo do tipo "diff"

http://en.wikipedia.org/wiki/Plagiarism_detection **-** *Página wiki* sobre deteção de plágio com secção sobre documentos de texto