

Examinamos los algoritmos que usa la herramienta diff

¿DÓNDE ESTÁ LA DIFERENCIA?

Diff es capaz de encontrar las diferencias entre dos versiones de un archivo. Os mostramos cómo encuentra los cambios y coincidencias de archivos sin afectar a los recursos del sistema. **POR ANDREAS ROMEYKE**

Para el usuario de la línea de comandos, descubrir las diferencias entre dos archivos de texto es fácil: un simple comando como `diff Version_1.txt Version_2.txt` es todo lo que necesitamos. No obstante, si hacemos un análisis más detenido, veremos que diff necesita una gran cantidad de memoria y algunos ingeniosos algoritmos para comparar archivos.

Este artículo investiga cómo consigue diff encontrar los cambios y coincidencias en archivos de muchos megabytes sin perjudicar los recursos del sistema.

Distancia de Edición

Cada cadena puede cambiarse por otra mediante inserción, borrado o reemplazamiento de caracteres individuales.

Un método posible para convertir *tier* en *tor* sería llevar a cabo los siguientes cambios: *tier* -> *ter* -> *tr* -> *tor*. Aunque una solución alternativa con menos pasos intermedios sería: *tier* -> *ter* -> *tor*.

El menor número de pasos necesarios para hacer el cambio nos proporciona una métrica para evaluar el grado de similitud entre dos cadenas. Esta métrica es conocida como la distancia de Levenshtein o distancia de edición, y es la base fundamental del método que usa diff para hacer cambios.

En aplicaciones prácticas, la mayor parte del archivo no presenta cambios en las comparaciones. Por tanto, el primer paso es excluir las partes idénticas.

	t	i	e	r
t	0	11	15	2
o	5	6	10	3
r	2	9	13	0

Figura 1: La visualización de la matriz revela las coincidencias (ceros), incluso si la posición de los caracteres ha cambiado entre las dos filas.

	o	t	t	e	r
l	3	8	8	7	6
o	0	5	5	10	3
t	5	0	0	15	2
t	5	0	0	15	2
o	0	5	5	10	3

Figura 2: Los fragmentos coincidentes se muestran como diagonales de ceros que discurren en paralelo a largo de la diagonal central de la matriz.

Para descubrir los cambios, incluso si se han desplazado respecto del original, necesitamos organizar el texto en una matriz como se muestra en la Figura 1.

Los números de la tabla hacen referencia a las diferencias entre los valores

	t	i	e	r
t	0	11	15	2
e	15	4	0	17
i	11	0	4	13
r	2	13	17	0

Figura 3: Las conmutaciones se muestran como interrupciones en diagonales de ceros. Los caracteres que se han conmutado se localizan con la línea que forma 90 grados respecto a la diagonal.

byte de cada carácter. Por tanto, un cero representa un carácter que no ha cambiado. A la coincidencia de mayor tamaño se le llama subsecuencia común mayor o LCS.

La distancia de edición se puede derivar de la longitud de la LCS aplicando la siguiente fórmula: $d(X, Y) = n + m$

$- 2 * |LCS(X, Y)|$, siendo $X = x_1...x_n$ y $Y = y_1...y_m$.

En una matriz así, los desplazamientos son fáciles de detectar: comparando *otter* con *lotto* (véase la Figura 2) los ceros (coincidencias) se localizan en una diagonal de la matriz (la diagonal que desciende de izquierda a derecha).

Listado 1: Búsqueda del LCS

```
01 sub lcs {
02   my $refmatrix=shift;
03   my $refxlst=shift;
04   my $refylst=shift;
05   my $m=scalar @$refxlst-1;
06   my $n=scalar @$refylst-1;
07   foreach my $i (1 .. $m) {
08     foreach my $j (1 ..
09       $n) {
10       if ($refxlst->[$i]
11         eq $refylst->[$j]) {
12         $refmatrix->[$i]->[$j] =
13           $refmatrix->[$i-1]->[$j-1]+1;
14       } elsif
15         ($refmatrix->[$i-1]->[$j] >=
16          $refmatrix->[$i]->[$j-1]) {
17         $refmatrix->[$i]->[$j] =
18           $refmatrix->[$i-1]->[$j];
19       } else {
20         $refmatrix->[$i]->[$j] =
21           $refmatrix->[$i]->[$j-1];
22       }
23     }
24   }
25   return $refmatrix;
26 }
```

Listado 2: Backtracking

```
01 # run backtracking on
02   lcs-matrix
03   sub backtracking_lcs {
04     my $refmatrix=shift;
05     my $ref_xlst=shift;
06     my $ref_ylst=shift;
07     my @lcs;
08     my $x=scalar @$ref_xlst
09       -1;
10     my $y=scalar @$ref_ylst
11       -1;
12     while ($y>0 && $x>0) {
13       my
14         $actual_value=$refmatrix->[$x]
15         ->[$y];
16       my $actual_x=$x;
17       if (
18         ($refmatrix->[$x-1]->[$y-1] >=
19          $refmatrix->[$x-1]->[$y]) &&
20         ($refmatrix->[$x-1]->[$y-1] >=
21          $refmatrix->[$x]->[$y-1])
22         ) { # go left upper
23         $x--; $y--;
24       } elsif
25         ($refmatrix->[$x-1]->[$y] >=
26          $refmatrix->[$x]->[$y-1]) { #
27         go left
28         $x--;
29       } else { # go upper
30         $y--;
31       }
32     }
33     @lcs=reverse @lcs; #
34     reverse because backtracking
35     return \@lcs;
36 }
37 # print out lcs matrix
38 sub print_lcs {
39   my $ref_matrix=shift;
40   my $ref_xlst=shift;
41   my $ref_ylst=shift;
42   print "LCS: ";
43   foreach my $i (@{
44     backtracking_lcs($ref_matrix,
45       $ref_xlst, $ref_ylst) }) {
46     print $ref_xlst->[$i];
47   }
48   print "\n";
49 }
```

	l	a	g	e	r
r	6	17	11	13	0
e	7	4	2	0	13
g	5	6	0	2	11
a	11	0	6	4	17
l	0	11	5	7	6

Figura 4: Los palíndromos (orden inverso en los caracteres) se muestran como diagonales en la matriz que suben desde la izquierda a la derecha.

Las conmutaciones (de las palabras de *teir* a *tier* del ejemplo, véase la Figura 3) se muestran como interrupciones en la matriz con ceros formando un ángulo de 90 grados con respecto a la diagonal principal a partir de sus centros.

Los palíndromos (orden inverso en los caracteres) se revelan como secuencias de ceros que descienden desde la esquina derecha a la esquina izquierda (la diagonal secundaria) en la matriz (véase la Figura 4).

		t	i	e	r
	0	0	0	0	0
t	0	1	1	1	1
e	0	1	1	2	2
e	0	1	1	2	2
r	0	1	1	2	3

Figura 5: En lugar de introducir las diferencias entre los valores de carácter, es más eficiente escribir la longitud de las subsecuencias en el parseo inicial.

Optimización del Tiempo de Ejecución

El tamaño de la matriz depende de la longitud de los textos. Si tenemos dos archivos de 10 KB, el número de compa-

Listado 3: Algoritmo Diff

```

01 # run backtracking on
    lcs-matrix
02 sub backtracking_lcs {
03     my $refmatrix=shift;
04     my $ref_xlst=shift;
05     my $ref_ylst=shift;
06     my @lcs;
07     my $x=scalar @$ref_xlst -1;
08     my $y=scalar @$ref_ylst -1;
09     while ($y>0 && $x>0) {
10         my
            $actual_value=$refmatrix->[$x]
            ->[$y];
11         my $actual_x=$x;
12         my $actual_y=$y;
13         my $actual_direction;
14         if (
15             ($refmatrix->[$x-1]->[$y-1] >=
            $refmatrix->[$x-1]->[$y]) &&
16             ($refmatrix->[$x-1]->[$y-1] >=
            $refmatrix->[$x]->[$y-1])
17         ) { # go left upper
18             $x-; $y-;
19             $actual_direction="ul";
20         } elsif
            ($refmatrix->[$x-1]->[$y] >=
            $refmatrix->[$x]->[$y-1]) { #
            go left
22             $x-;
23             $actual_direction="l";
24         } else { # go upper
25             $y-;
26             $actual_direction="u";
27         }
28         # check if value is
            changed, then push to @lcs
29         if ($actual_value >
            $refmatrix->[$x]->[$y]) {
30             # push @lcs, $actual_x;
31             push @lcs,
                (".$ref_xlst->[$actual_x].")
                ;
32         } else {
33             if ($actual_direction eq
                "u") {
34                 push @lcs,
                    (".$ref_ylst->[$actual_y].")
                    ;
35             } elsif
                ($actual_direction eq "l") {
36                 push @lcs,
                    (".$ref_xlst->[$actual_x].")
                    ;
37             } else {
38                 push @lcs,
                    (".$ref_ylst->[$actual_y].")
                    ;
39                 push @lcs,
                    (".$ref_xlst->[$actual_x].")
                    ;
40             }
41         }
42     }
43     while ($y > 0) { # get last
            stuff of ylst
44         push @lcs,
            (".$ref_ylst->[$y].");
45         $y-;
46     }
47     while ($x > 0) { # get last
            stuff of xlst
48         push @lcs,
            (".$ref_xlst->[$x].");
49         $x-;
50     }
51     @lcs=reverse @lcs; # reverse
            because backtracking
52     return \@lcs;
53 }
54 # print out lcs matrix
55 sub print_diff {
56     my $ref_matrix=shift;
57     my $ref_xlst=shift;
58     my $ref_ylst=shift;
59     print "DIFF: ";
60     foreach my $i (@{
            backtracking_lcs($ref_matrix,
            $ref_xlst, $ref_ylst) }) {
61         print $i;
62     }
63     print "\n";
64 }

```

raciones es sorprendentemente alto: $10000 * 10000 = 100000000$, lo que significa que necesitamos 100 MB de RAM sólo para guardar la matriz. La búsqueda de las coincidencias requiere más memoria.

Un proceso computacional que calcula valores múltiples veces se puede optimizar. La Programación Dinámica (véase el cuadro "Programación Dinámica") reduce el consumo de memoria y ahorra tiempo de computación.

Además consigue mantener bajo el número de comparaciones cuando se comparan dos versiones de un texto en una matriz: en lugar de diferencias a nivel de bits entre dos caracteres, la matriz mostrada en la Figura 5 guarda el número de caracteres coincidentes desde el comienzo de la cadena. El Listado 1 proporciona el código Perl usado para implementar este método.

Usando los valores mostrados en la Figura 5, un algoritmo de backtracking puede determinar rápidamente la subsecuencia mayor común de la cadena:

1. Comenzamos por el valor máximo. Seleccionamos la entrada mayor arriba y a la izquierda, o a la izquierda, o arriba de la posición actual.

2. En caso de empate, tomamos el camino en diagonal.

3. Recorremos la matriz. El LCS se halla si existen varias entradas con el mismo valor máximo.

La Figura 6 muestra el camino resultado de este algoritmo en la matriz. El Listado 2 implementa este mismo algoritmo en Perl. Para permitirle al script que termine adecua-

damente, la cadena debe contener una secuencia de valores nulos al comienzo, como se muestra en la figura.

No habrá que añadir mucho al algoritmo visto en la anterior sección para generar las diferencias entre dos archivos o cadenas tal y como lo hace diff. Cada vez que el camino seguido por la matriz cambia de dirección hacia arriba o a la izquierda, se ha borrado un carácter o se ha insertado en la versión nueva.

El script del Listado 3 detecta estos cambios. El bucle *while* de las líneas 43 a 47 aseguran que el algoritmo toma en consideración los caracteres representados por ceros en la matriz.

Aunque la programación dinámica evita numerosos cálculos, los desarrolladores de la herramienta diff de Unix (posteriormente conocidas como las diff-utils, [1]) tuvieron que sacarse otra carta de debajo de la manga.

La herramienta diff se diseñó fundamentalmente para usarla con código fuente. Para poder controlar el enorme tamaño de este tipo de archivos con la memoria de los ordenadores de los años 80, diff no compara letra a letra, sino línea a línea.

Para ello, el programa calcula en primer lugar un hash para cada línea, antes de calcular las diferencias entre los hashes en un segundo paso. No necesita comparar las líneas letra a letra si los hashes son idénticos. Este método ahorra una enorme cantidad de memoria.

En 1986, Eugene Myers desarrolló un veloz algoritmo que es la base de las populares diff-utils [6]. Las herramientas alternativas a diff basadas en interfaz gráfica, como Meld [7] o KDE Kompare [8], se basan todas en este método. De hecho, a pesar de sus bonitos gráficos, Kompare confía en el fondo en la veterana herramienta diff.

Más Aplicaciones

La técnica que usa diff, además de descubrir diferencias, su algoritmo puede usarse para encontrar también las coincidencias, y por tanto probar qué cierto código ha sido reutilizado. Para proyectos de gran tamaño, la aparición de muchos duplicados en el código es prueba de un reaprovechamiento exitoso. Una variante del algoritmo de diff es incluso capaz de comparar notas musicales tocadas con las notas que se supone debía tocar.

		t	i	e	r
	0	0	0	0	0
t	0	1	1	1	1
e	0	1	1	2	2
e	0	1	1	2	2
r	0	1	1	2	3

Figura 6: Para descubrir la mayor subsecuencia, comenzamos en el valor máximo de la tabla y caminamos hacia atrás por la matriz, usando un sencillo algoritmo.

Si la matriz de distancias (véase la Figura 4) muestra la diferencia entre las pulsaciones de un teclado de ordenador (lo que se conoce como distancia de teclado), en lugar de la diferencia entre los códigos del carácter, se puede aplicar a palabras tecleadas incorrectamente para averiguar lo que una persona quería teclear. Una aplicación interesante de diff se encuentra en biología, donde se usa para secuenciar y catalogar genes. ■

RECURSOS

- [1] Manual de las GNU Diffutils, 2002: <http://www.gnu.org/software/diffutils/manual/diff.html>
- [2] Darren C. Atkinson y William G. Griswold, "Effective pattern matching of source code using abstract syntax patterns: Software Practice and Experience", 36 (4), p. 413-447, 2006.
- [3] K. Nandan Babu y Sanjeev Saxena, "Parallel algorithms for the longest common subsequence problem", Enero 1999.
- [4] J. W. Hunt y M. D. McIlroy "An algorithm for differential file comparison: Technical Report" CSTR 41, Laboratorios Bell, Murray Hill, NJ, 1976.
- [5] Moritz G. Maaß, "Matching statistics: efficient computation and a new practical algorithm for the multiple common substring problem: Software Practice and Experience", 36 (3), p. 305-331, Marzo 2006.
- [6] E. W. Myers, "An o(ND) difference algorithm and its variations: Algorithmica, 1 (2)", p. 251-266, 1986; <http://citeseer.ist.psu.edu/myers86ond.html>
- [7] Meld: <http://meld.sourceforge.net>
- [8] Kompare: <http://www.caffeinated.me.uk/kompare>

Programación Dinámica

La programación dinámica supone un importante concepto en las ciencias computacionales, y es también el mejor método para resolver problemas de optimización. En muchos casos, es más fácil dividir el problema, resolver las subtarefas de manera individual, y usar los resultados en pasos sucesivos.

El cálculo de potencias es un sencillo ejemplo que se remonta a los tiempos en los que los recursos para computación eran escasos: para calcular la octava potencia de un número podemos dividir el cálculo $n * n * n * n * n * n * n * n$ en los pasos intermedios $((n * n) * (n * n)) * ((n * n) * (n * n))$. Si guardamos los resultados de $(n * n)$ y $((n * n) * (n * n))$ de manera temporal, sólo necesitamos tres multiplicaciones en lugar de siete.