Tarea 2: Calculo Recursivo de Distancia de Edición

Sebastián Jesús Sanhueza Bustamante

June 10, 2024

1 Casos de Prueba

Para los casos de prueba se elegieron las siguientes cuatro palabras:

- **sofia**
- **sebastian**
- **pocima**
- **pescar**

se crearon los siguientes tests utilizando distintas combinaciones de pares ($source, target$) de las palabras seleccionadas anteriormente, junto su cantidad minima de operaciones para transformar la string *source* a la string *target*:

source	*target*	operaciones minimas	justificación
sebastian	sofia	6	”s” se mantiene, ”ebast” se le elimina 3 caracteres y se reemplazan 2 por ”o” y ”f”, ”ia” se mantiene y ”n” se elimina.
sebastian	pescar	6	”s” se reemplaza por ”p”, ”e” se mantiene, ”ba” se elimina, ”s” se mantiene, ”ti” se le elimina 1 caracter y se reemplaza otro por ”c”, y ”n” se reemplaza por ”r”.
sebastian	pocima	8	”sebast” se eliminan 3 caracteres y se reemplazan los otros 3 por ”poc”, ”i” se mantiene y se reemplaza ”an” por ”ma”.
sofia	sebastian	-1	al no poseer una operación de inserción no es posible transformar de sofia a sebastian
sofia	pocima	-1	al no poseer una operación de inserción no es posible transformar de sofia a pocima
sofia	pescar	-1	al no poseer una operación de inserción no es posible transformar de sofia a pescar
pocima	sofia	3	se reemplaza la ”p” por ”s”, se mantiene la ”o”, se reemplaza la ”c” por ”f”, se mantiene la ”i” y se elimina ”m”.
pocima	sebastian	-1	al no poseer una operación de inserción no es posible transformar de pocima a sebastian
pocima	pescar	5	se mantiene la ”p”, se reemplaza ”ocima” por ”escar”.
pescar	pocima	5	se mantiene la ”p”, se reemplaza ”escar” por ”ocima”.
pescar	sofia	5	se reemplaza ”pesc” por ”sofi”, se mantiene la ”a” y se elimina la ”r”.
pescar	sebastian	-1	al no poseer una operación de inserción no es posible transformar de pescar a sebastian

2 Formula Recursiva

Algorithm 1 Edit distance con Delete y Replace.

procedure EDIT_DISTANCE($source, target, first\_call = true$)
  $i \leftarrow$ Length($source$)
  $j \leftarrow$ Length($target$)
  if $first\_call$ is $true$ and $i < j$ **then**
    return -1
  end if
  if i or j is 0 **then**
    return $i + j$
  end if
  if $source[i-1]$ equal to $target[j-1]$ **then**
    return EDIT_DISTANCE($source.subString(0, i-1), target.subString(0, j-1,), false$)
  else
    $Delete \leftarrow$ EDIT_DISTANCE($source.subString(0, i-1), target, false$)
    $Replace \leftarrow$ EDIT_DISTANCE($source.subString(0, i-1), target.subString(0, j-1), false$)
    $min\_value \leftarrow$ Min($Delete, Replace$) + 1
    return min_value
  end if
end procedure

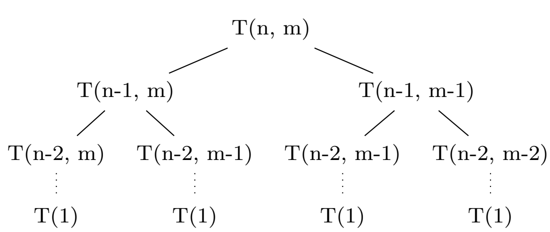
3 Implementación

La implementación de los algoritmos se realizo utilizando Python y se encuentra en el siguiente repositorio de GitHub.
`https://github.com/sanhue903/FEDA-Tarea-2`

4 Complejidad

4.1 Algoritmo sin Memoización

- **Complejidad Temporal**
  Para el calculo de la complejidad temporal del algoritmo sin memoización podemos utilizar el peor caso del algoritmo para obtener la altura maxima del arbol recursivo. Ya que el caso base se llega cuando el tamaño de una de las dos string que se dan como parametro es igual a 0, como en esta versión de "edit distance" no existe la operación Insert, la altura maxima del arbol de recursión depende del tamaño de la string *source* por lo que la altura de este arbol es n.



  Como en cada llamada solamente se realizan un par de comparaciones, podemos decir un nodo tiene como coste c, y como es un arbol binario cada nivel tendra la doble cantidad de nodos que el nivel anterior, teniendo asi el coste total de una altura i es igual a 2^i, con $0 \leq i \leq n$, ya con esta información podemos obtener el costo total de esta recursión realizando la siguiente sumatoria.

$$c * \sum_{i=0}^{n} 2^i = c * \frac{1 - 2^{n-1}}{1 - 2}$$

Como resultado de la sumatoria podemos decir que que para esta versión de "edit distance" su complejidad temporal es $O(2^n)$.

- **Complejidad Espacial**
  Como en el stack de memoria se guardan las funciones que todavia no terminan, se acumulan ahi hasta que una función llegue al caso base, para esto es necesario recorrer la string *source* por completo, por lo que su complejidad espacial es $O(n)$.

4.2 Algoritmo con Memoización

- **Complejidad Temporal**
  Como se esta utilizando memoización, los subproblemas que ya fueron resueltos pueden ser consultados en el cache, por lo que se tiene que calcular la cantidad de subproblemas nuevos que se tiene que solucionar, para calcular esta cantidad debemos obtener el numero de posibles combinaciones de indices i y j, con $0 \leq i \leq n$ y $j \leq i$, podemos utilizar la siguiente expresión:

$$c * \binom{n}{2} = c * \frac{n!}{2! * (n-2)!} = c * \frac{n * (n-1)}{2}$$

  Por lo que la complejidad temporal de este algoritmo es $O(n^2)$.

- **Complejidad Espacial**
  La complejidad espacial de este algoritmo es fuertemente influenciado por el cache utilizado para la memoización, como el cache es una matriz de $n * m$, con $m \leq n$, se tiene que la complejidad espacial de este algoritmo es $O(n^2)$.

5 Protocolo Experimental

Para realizar la experimentación, se lee un archivo .txt con las frases a utilizar, luego se prueba el algoritmo sin memoización, el algoritmo se repetira una x cantidad de veces para cada combinación de frases posible, tomando el tiempo de ejecución promedio para cada una de estas combinaciones, se vuelve a repetir el mismo proceso para el algoritmo con memoización.

6 Experimentación

cabe aclarar que los resultados de 0ms para ambos algoritmos de debe al caso base utilizado cuando una string *target* es más grande que la string *source*.

Tamaños	Tiempo no memoización(ms)	Tiempo con memoización(ms)
13 - 19	0	0
13 - 16	0	0
13 - 25	0	0
19 - 13	206.8	0.1
19 - 16	202.4	0.1
19 - 25	0	0
16 - 13	13.6	0.1
16 - 19	0	0
16 - 25	0	0
25 - 13	3265.7	0.2
25 - 19	9231.0	0.5
25 - 16	4801.9	0.2

Table 1: Resultados del experimento

7 Conclusión

Se puede observar claramente como los tiempos de ejecución varian de forma muy distinta para cada algoritmo, mientras que el algoritmo sin memoización pasamos de tiempos cercanos a los 13ms para las combinación con menos caracteres, pasa a llegar a más de 9000ms para la combinación de strings más grande, en cambio los tiempos de ejecución para el algoritmo con memoización no superaron los 1ms de ejecución para ningun caso de prueba, demostrando asi la diferencia de complejidades temporales que tienen cada uno.