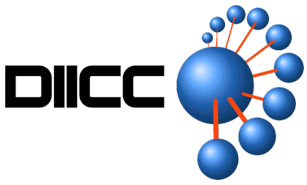


FUNDAMENTOS DE ESTRUCTURAS DE DATOS Y ANÁLISIS DE ALGORITMOS

INFORME N°4

"Programación Dinámica"

Julio 2023

ERICH GERMÁN GRÜTTNER DÍAZ

1. Introducción

Este informe tiene por objetivo analizar soluciones al problema DELETE INSERT EDIT DISTANCE, que consiste, básicamente, en encontrar la distancia desde una cadena de texto S, de tamaño n, hacia otra cadena T, de tamaño m, utilizando la mínima cantidad de operaciones Delete e Insert. [1]

Se analizan dos soluciones, una clásica y otra “adaptativa”. Para la primera se utiliza el algoritmo de programación dinámica Wagner-Fischer sin considerar la operación “REPLACE”, mientras que para la segunda opción se construye un algoritmo propio. [4]

La implementación de ambos algoritmos a C++ se realiza de manera similar a los informes anteriores, aprovechando las herramientas y librerías disponibles para la medición de rendimiento.

Las pruebas a realizar corresponden a comparaciones entre textos del proyecto Gutenberg [7] utilizando ambos algoritmos, en una primera instancia sin cambios y en una segunda fase realizando cambios acotados e incrementales.

En el apartado de resultados experimentales, se muestran datos y gráficos resultantes de la ejecución de las diversas tareas en el software. Para ello se utiliza la librería “Chrono” de C++. En tanto que los gráficos se obtienen a través de Python usando la librería “Matplotlib”. [8]

Finalmente se presentan las conclusiones, que permiten observar el rendimiento de ambas soluciones en escenarios diferentes como el volumen de las palabras a comparar y la cantidad de diferencia entre ellas.

Este informe se complementa con un repositorio Github [9] en donde se puede encontrar tanto el código utilizado en los diferentes experimentos, como los diferentes archivos de salida y gráficos de análisis correspondientes.

2. Descripción del problema

En ciencias de la computación, “edit distance” es una forma de cuantificar cuan diferentes entre ellos son dos strings. Esto se puede hacer contando el mínimo número de operaciones requeridas para transformar un string en otro.

Este tipo de problemática resulta útil en el procesamiento del lenguaje natural, y sus aplicaciones van desde la corrección ortográfica hasta la comparación de secuencias de DNA.

Formalmente, dados dos strings a y b y un alfabeto Σ (ej: caracteres ASCII), la “edit distance” $d(a,b)$ es la mínima (de menor peso) serie de operaciones de edición que transforman a en b. Uno de los sets de operaciones de edición más sencillos fue definido por Levenshtein en 1966: INSERT, DELETE y SUBSTITUTION. Cada una de estas operaciones, según su definición original, tiene un costo unitario (excepto la sustitución de un carácter por sí mismo, que tiene cero costo).

También existen otras tipos de “edit distance” tales como “Hamming distance” que solo permite substitución y solamente aplica para cadenas del mismo largo. O también se puede mencionar “Damerau-Levenshtein distance” que además incorpora la transposición de dos caracteres adyacentes.

Para el presente informe se solicitó enfocar el estudio en una solución clásica y otra adaptativa. Además de solamente considerar las operaciones Delete e Insert.

2.1 Solución clásica:

Algoritmo 1: Clásica – Basado en el algoritmo Wagner - Fischer

```
1:  Function CLASICA (S, T) is
2:    m ← S.size();
3:    n ← T.size();
4:    distancia [i][0] ← i ∀ i ∈ 0..m;
5:    distancia [0][j] ← j ∀ j ∈ 0..n;
6:    for i ← 1 to m do
7:      for j ← 1 to n do
8:        if S [i - 1] == T [j – 1] then
9:            distancia [i][j] ← distancia [i -1][j -1];
10:       else
11:           distancia [i][j] ← mínimo(
12:           distancia [i – 1][j] + 1,       //DELETE
13:           distancia [i][ j - 1] + 1       //INSERT
14:            );
15:       end
16:     end
17:   end
18:   return distancia [m][n];
19: end
```

Esta solución se basa en el algoritmo de programación dinámica [3] de Wagner-Fischer, que indica que si se utiliza una matriz que contenga las "edit distance" entre los prefijos del primer string y todos los prefijos de la segunda, entonces es posible calcular los valores de la matriz a través de "Flood filling" (Algoritmo de relleno por difusión [5]), encontrando la distancia en el último valor calculado. Así, cada celda representa un sub-problema.

	""	C	O	S	A
""	0	1	2	3	4
C	1	0	1	2	3
A	2	1	2	3	2
S	3	2	3	2	3
A	4	3	4	3	2

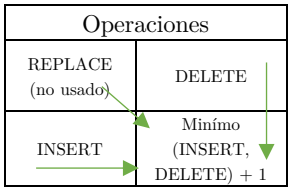


Figura 1 – Ejemplo cálculo distancia y operaciones

El "Flood filling" se va realizando de acuerdo a las operaciones disponibles, en este caso DELETE e INSERT.

Este algoritmo tiene una complejidad dentro de O(mn), donde m = largo de palabra 1 y n = largo de palabra 2. Y el costo espacial está dentro de n+m+O(1).

En rigor el orden espacial de la función es **clásica** es m*n + O(1), ya que se utiliza una matriz de tamaño (m + 1)(n+1) para almacenar los cálculos de distancia, lo que ocupa espacio proporcional a m*n, y el espacio adicional utilizado es constante.

Existe una variación que solamente utiliza un vector en vez de una matriz, consiguiendo complejidad espacial de O(m), ya que para el cálculo solamente se requiere que sea almacenada la fila anterior y la actual. El costo de esta solución es que se no se obtiene el flujo completo de la transformación entre una palabra y otra (camino en amarillo de la figura 1).

2.2 Solución adaptativa

Esta solución requiere de dos funciones: Verifica y Adaptativa, que se describen a continuación:

Algoritmo 2: Verifica

```
1:  Function VERIFICA (S, T, D) is
2:    m ← S.size();
3:    n ← T.size();
4:    if (abs(m-n) > D) then
5:       return m + n + 1;
6:    end
7:
8:    for i ← 0 to m do
9:       distancia[i][max(0, i – D – 1) ← i;
10:   end
11:
12:   for j ← 0 to n do
13:      distancia[max(0, j – D – 1)][j] ← j;
14:   end
15:
16:
17:
18:   for i ← 1 to m do
19:     for j ← max(1,i-D) to min(n,i+D) do
20:        if S[i - 1] == T[j – 1] then
21:           distancia[i][j] ← distancia [i – 1][j – 1];
22:        else
23:           distancia[i][j] ← 1+ mínimo(
24:           distancia[i – 1][j], //DELETE
25:           distancia[i][j – 1]  //INSERT
26:           );
27:        end
28:     end
29:   end
30:   if distancia[m][n] <= D then
31:     return distancia[m][n];
32:   end
33:   return (m + n + 1);
34: end
```

Algoritmo 3: Adaptativa

```
1:  Function ADAPTATIVA (S, T) is
2:    m ← S.size();
3:    n ← T.size();
4:    distancia_llamada ← 0;
5:    aux ← 0;
6:    while (distancia_llamada <= m + n)
7:    do
8:       resultado_verifica = verifica(S, T, distancia_llamada);
9:       if (resultado_verifica < m + n + 1) then
10:         return resultado_verifica;
11:      end
12:      if (distancia_llamada * 2 > m + n ) then
13:         distancia_llamada ← m + n;
14:      else
15:         distancia_llamada ← 2^aux;
16:         aux ← aux + 1;
17:      end
18:   end while
      return m + n;
    end
```

En esencia, este algoritmo se basa en el cálculo clásico, pero utilizando el método de "sondeo" para alcanzar más rápidamente una solución. Se verá que esto aplica principalmente cuando la diferencia entre strings no es muy grande.

La motivación para esa solución es que, en el caso de palabras no muy diferentes, se puede observar de que toda la acción de cómputo ocurre en el pasillo central de la matriz, por ende el resto de cálculos podrían ser omitidos, ganando en velocidad.

La función **adaptativa** utiliza la función **verifica** para determinar la distancia de edición mínima entre S y T. Comienza con una distancia de llamada inicial de 0 y llama a "verifica" con incrementos de distancia de llamada hasta que se encuentre una distancia de edición mínima menor que m + n + 1 (la longitud total de las cadenas). Si la distancia de llamada se duplica y aún no se encuentra una distancia de edición mínima menor, se establece la distancia de llamada como m + n (la máxima posible). Finalmente, se devuelve la distancia de edición mínima encontrada o m + n si no se encuentra ninguna distancia menor.

El objetivo de la función **verifica** es determinar si la diferencia entre longitudes de los strings S y T es mayor que un D dado… si es así, se devuelve la suma de las longitudes de ambas cadenas más uno (m + n + 1). De lo contrario, se calcula la distancia de edición mínima entre las dos cadenas utilizando el algoritmo de programación dinámica. La distancia de edición mínima se almacena en una matriz dp, donde dp[i][j] representa la distancia entre los primeros i caracteres de S y los primeros j caracteres de T.

La complejidad de tiempo de **verifica** es O((D+1)(n+m)), ya que los bloques anidados recorren un rango de D+1 elementos alrededor de la diagonal principal de la matriz de cálculo, y, en el peor caso, este rango puede abarcar tanto la longitud de S como la longitud de T.

Sin embargo, al tener una validación de que la diferencia absoluta entre m y n debe ser menor a D (lo que retorna como resultado m+n+1), puede hacer que la ejecución de termine en forma abrupta.

La complejidad de espacio de **verifica** es m*n+O(1), ya que se utiliza el mismo tamaño de matriz que en la clásica, de (m+1)(n+1) ocupando espacio proporcional m*n, y el espacio adicional utilizado es constante.

La complejidad de la función **adaptativa** es de O((d+1)(n+m)), ya que depende del rendimiento de **verifica**. Las llamadas crecientes (o “gallop”) son en base 2, pero pueden detenerse abruptamente por el comportamiento de **verifica**.

Este algoritmo se desarrolló en conjunto con los estudiantes del Doctorado de Ciencias de la Computación de la Universidad de Concepción.

3. Implementación de las soluciones

3.1 Equipo de pruebas

Para la realización de las pruebas se utilizó un equipo MacbookPro con procesador M1 y 8Gb de memoria. El chip M1 tiene 8 núcleos (4 de alta eficiencia a 3.2 GHz + 4 de alto rendimiento a 2.0 GHz) y una velocidad de transferencia de 50Gb por segundo.

Los núcleos de alto performance tienen un caché de instrucciones L1 de 192 KB, un caché de datos L1 de 128 KB, y comparten un caché L2 de 12 MB. Mientras que los núcleos de alta eficiencia energética tienen caché de instrucciones L1 de 128 KB, un caché de datos L1 de 64KB y comparten un caché L2 de 4MB. El SoC también tiene un caché a nivel de sistema, de 8MB que es compartido por el GPU.

3.2 Prueba 1: entre dos textos del proyecto Gutemberg

Se seleccionaron los libros “Alice’s Adventures in Wonderland”, de Lewis Carroll (3760 líneas) [10] y “Metarmorphosis”, de Frank Kafka [11], con 2267 líneas.

Inicialmente se intentó comparar los libros completos, pero dado el alto costo espacial (matriz m*n), que se traduce en requerimiento de memoria RAM, no pudo ser realizado.

Por lo tanto se dividieron los libros en 5 bloques proporcionales a su tamaño total. Cada ejecución tomó una versión de cada libro y procesó la distancia utilizando ambos algoritmos. El dataset está disponible en [12]

3.3 Prueba 2: entre un texto del proyecto Gutenberg y alteraciones menores en ese texto

Se utilizó como base el libro “Alice’s Adventures in Wonderland”, reducido a 100 líneas. Luego se prepararon 20 copias con el siguiente cambio: insertar un espacio y cambiar un letra. El cambio se realizó de manera incremental. El dataset está disponible en [12]

3.4 Medición de tiempos de ejecución y generación de gráficos

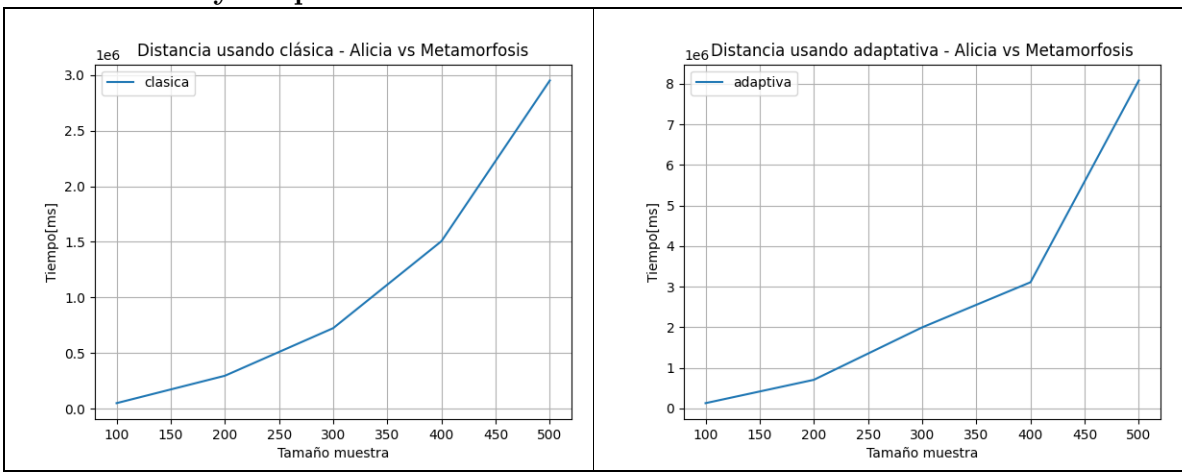
Se utilizó la librería “Chrono” de STL, para medir la ejecución del código. Cada tarea ejecutada fue medida en base a un promedio de ejecución de 10 intentos, y almacenada en un archivo CSV con los tiempos para cada tamaño de dataset.

Se realizó en base a **make**. Adicionalmente, se incorporó un archivo de tipo Bash para ejecutar otras tareas como limpieza y carga del programa.

Finalmente se utilizó un script en lenguaje Python, que utiliza la librería **MatPlotLib** [8] y que permite generar gráficos customizados.

4. Resultados experimentales

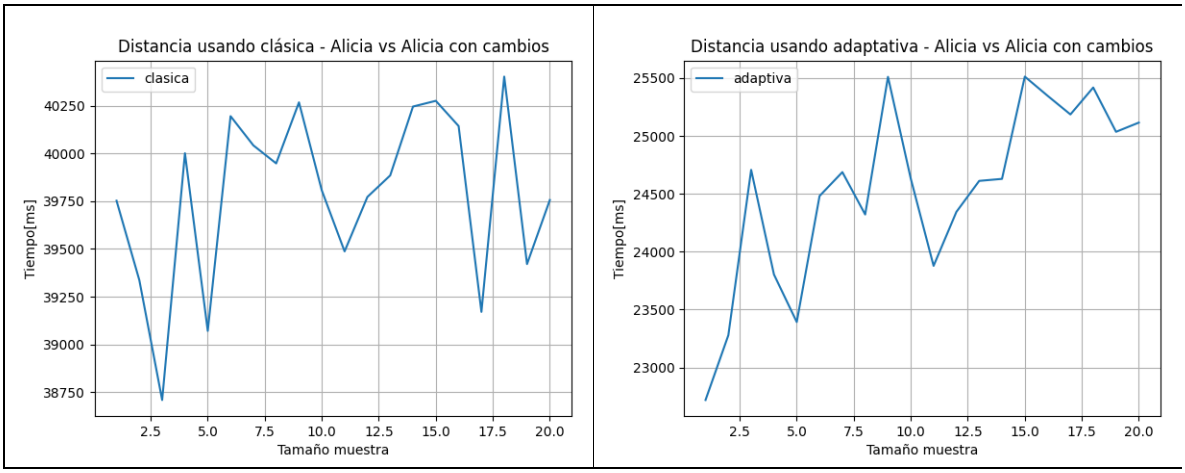
4.1 Clásica y adaptativa – “Alicia” vs “Metarmorfosis”



Si bien se observan curvas similares a un desempeño dentro de O(n*m), la diferencia global de tiempos es de casi 3 veces mejor para la función clásica.

Tamaño estructuras

4.2 Clásica y adaptativa – “Alicia” vs “Alicia” con modificaciones incrementales leves



Si bien no es posible determinar una curva de tendencia a priori, sí es notable la diferencia de tiempo hacia el rendimiento de la función adaptativa, que en este escenario se muestra muy superior.

Ambas soluciones utilizan una matriz de enteros (int), de tamaño (m+1) x (n+1), donde m=largo string S y n=largo string T.

Inicialmente se intentó comparar ambos libros, “Alicia” (3758 líneas, 164015 caracteres) con “Metamorfosis” (2266 líneas, 138407 caracteres). Esta combinación requiere de una matriz de enteros de 164016 x 138407. Asumiendo que el tamaño del tipo de datos **int** es de 4 bytes:

- Tamaño en bytes = 164016 x 138407 x 4 = 905,093,275,904 bytes
- Tamaño en megabytes = 863,286.8 MB
- Tamaño en gigabytes = 842.2 GB

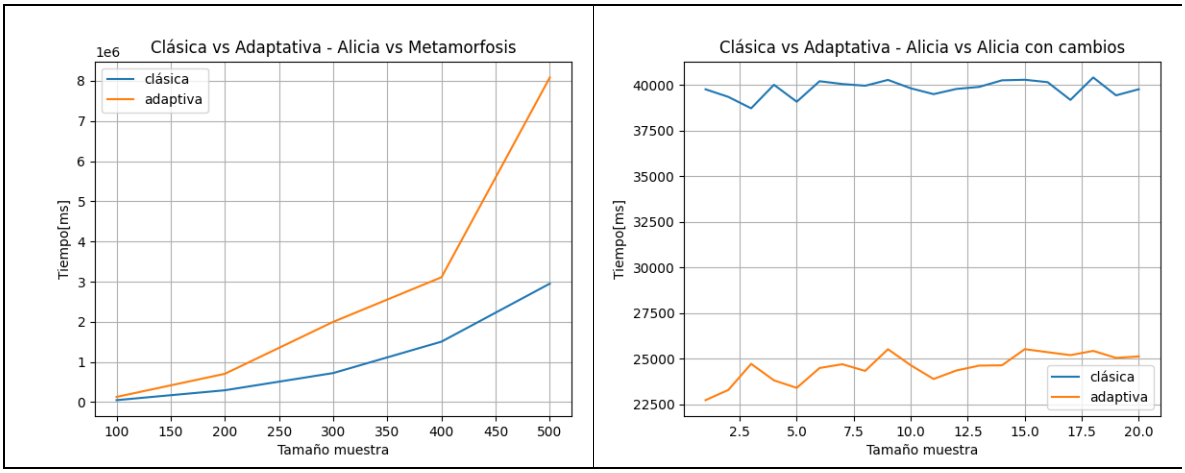
Estos requerimientos de tamaño hacen prácticamente imposible realizar el cálculo en un computador tradicional. Es por ello que se parcelaron los archivos en trozos más pequeños de hasta máximo 500 líneas: “Alicia” (25041 caracteres) y “Metamorfosis” (30042). Lo que genera una matriz de int de 25042 x 30043

- Tamaño en bytes = 25042 x 30043 x 4 = 3,002,545,368 bytes
- Tamaño en megabytes = 2,864.6 MB
- Tamaño en gigabytes = 2.8 GB

Que, evidentemente, es mucho más manejable.

Para el resto de las pruebas se utilizaron archivos más pequeños de 100 líneas.

5. Conclusiones



- Ambas soluciones presentan curvas similares al operar con tamaños de muestra grandes (manejables). En este escenario, el algoritmo **clásico** es superior.
- Al contrario, cuando las muestras son pequeñas y sobre todo cuando las diferencias entre los strings no son grandes (ya que fueron manipuladas para tal efecto), el algoritmo **adaptativo** presente un mejor rendimiento.
- Debido a la complejidad espacial de la solución **clásica** O(m*n), no fue posible procesar grandes tamaños de información debido a la cantidad de RAM necesaria para ello. Si bien la alternativa adaptativa optimiza el espacio, su alto tiempo de ejecución complica el procesamiento de grandes volúmenes de datos.
- Se investigó una solución que solamente utiliza un vector, lo que disminuye drásticamente su costo espacial de (m*n) a (m), pero con el costo de no almacenar los pasos utilizados para realizar los cálculos de distancia, funcionalidad que es aprovechada por la función adaptativa.
- Gracias a la programación dinámica es posible optimizar cálculos, utilizando el concepto de resolución de sub-problemas. Sin embargo, se observó que la solución clásica “gasta” tiempo de procesamiento que es posible aprovechar a través de una función adaptativa, concentrándose en donde, en general, ocurre mayormente la acción, o sea, el pasillo de la matriz.

6. Referencias

[1] Wikipedia, Edit Distance. [En línea]. Disponible: https://en.wikipedia.org/wiki/Edit_distance

[2] Wikipedia, Levenshtein Distance. [En línea]. Disponible: https://en.wikipedia.org/wiki/Levenshtein_distance

[3] Wikipedia, Programación Dinámica. [En línea]. Disponible: https://es.wikipedia.org/wiki/Programación_dinámica

[4] Wikipedia, Algoritmo Wagner-Fischer. [En línea]. Disponible: https://en.wikipedia.org/wiki/Wagner–Fischer_algorithm

[5] Wikipedia, Flood Fill. [En línea]. Disponible: https://en.wikipedia.org/wiki/Flood_fill

[6] Wikipedia, Dynamic Programming. [En línea]. Disponible: https://en.wikipedia.org/wiki/Dynamic_programming

[7] Gutenberg Project. [En línea]. Disponible: https://www.gutenberg.org

[8] Matplotlib. [En línea]. Disponible: https://matplotlib.org

[9] Github, Repositorio completo. [En línea]. Disponible: https://github.com/egruttner/FEDA-Tarea4/tree/main

[10] Gutenberg Project, “Alice’s Adventures in Wonderland”, Lewis Carroll. [En línea]. Disponible: https://www.gutenberg.org/ebooks/11

[11] Gutenberg Project, “Metamorphosis”, Franz Kafka. [En línea]. Disponible: https://www.gutenberg.org/ebooks/5200

[12] Github, Datasets. [En línea]. Disponible: https://github.com/egruttner/FEDA-Tarea4/tree/main/code/datasets