|  |  |  |
| --- | --- | --- |
|  |  |  |

FUNDAMENTOS DE ESTRUCTURAS DE DATOS

Y ANÁLISIS DE ALGORITMOS

**INFORME**

**“Proyecto Semestral”**

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]](#uno)

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]](#cuatro)

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]](#siete) 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]](#ocho)

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]](#nueve) 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. Estructuras de datos

Para ambas soluciones se cuenta con una vector de información común, que contiene la lista única de autores más una codificación entera propia.

## 2.1 Matriz de adyacencia

https://es.wikipedia.org/wiki/Matriz\_de\_adyacencia

En teoría de grafos y ciencias de la computación, una matriz de adyacencia es una matriz cuadrada utilizada para representar un grafo finito. Los elementos de la matriz indican que pares de vértices son adyacentes o no en el grafo.

Para el presente caso, cada vértice representará un autor y cada arista representará una relación de co-autoría entre trabajos académicos.

Normalmente se utiliza una matriz con 0 y 1 para indicar la presencia o ausencia de la relación (arista), pero en este caso se reemplazará el 1 por la cantidad de veces que han realizado, los autores, algún trabajo en conjunto.

Se describe de manera detalla la estructura de datos, incluyendo un ejemplo explicativo, algoritmo de construcción, operaciones, complejidades y referencias bibliográficas.

* Ejemplo de implementación
* El algoritmo de construcción fue el siguiente
* El algoritmo para buscar pares e impares
* Complejidad de tiempo para construcción
* Complejidad de tiempo para funciones de búsqueda

## 2.2 Lista de adyacencia

<https://en.wikipedia.org/wiki/Adjacency_list>

En teoría de grafos y en ciencia de la computación, una lista de adyacencia es una colección de listas desordenadas utilizadas para representar un grafo finito. Cada lista desordenada dentro de una lista de adyacencia describe el set de vecinos de un particular vértice en el grafo.

* Ejemplo de implementación
* El algoritmo de construcción fue el siguiente
* El algoritmo para buscar pares e impares
* Complejidad de tiempo para construcción
* Complejidad de tiempo para funciones de búsqueda

## 3. Implementación

Describir los detalles de la implementación realizada durante el desarrollo del proyecto. Se deben incluir decisiones técnicas importantes, como las métricas utilizadas para cada consulta y detalles de implementación de cada algoritmo.

* Proceso de extracción de datos XML
* Creación de índice de autores
* Proceso de carga de datos en matriz

## 4. Experimentación

Se debe incluir una sección experimental, donde se muestren los resultados experimentales de evaluar la implementación propuesta con el dataset.

Se utilizaron dos datasets: el archivo oficial dblp.xml (3,91 GB) y un dataset propio denominado test.xml (5 KB), que contiene extractos de información para realizar pruebas. Ambos datasets comparten el archivo de definición dblp.dtd.

Una vez realizada la carga en memoria del archivo, internamente se realizaron pseudo datasets, parcelando la carga en estructuras con parámetros limitantes. Por ejemplo: sólo se hicieron cargas de tags “asdasd” y “dsfsdfd”. Y también se limitó la carga por cantidad de registros.

El máximo de registros que se pudieron cargar con matriz fue

El máximo de registros que se pudieron cargar con lista fue

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

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

|  |  |
| --- | --- |
| **Algoritmo 3:** Adaptativa | |
| 1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13:  14:  15:  16:  17:  18: | **Function** ADAPTATIVA *(S, T)* **is**  *m* ← *S.size*();  *n* ← *T.size*();  *distancia\_llamada* ← 0;  *aux* ← 0;  **while (***distancia\_llamada* <= *m* + *n***)**  **do**  resultado\_verifica **= verifica(***S*, *T*, *distancia\_llamada***);**  **if** (*resultado\_verifica < m* + *n* + 1) **then**  **return** *resultado\_verifica;*  **end**  **if** (*distancia\_llamada* \*2 *> m* + *n* ) **then**  *distancia\_llamada* ← *m* + *n;*  **else**  *distancia\_llamada* ←2*aux;*  *aux* ← aux + 1;  **end**  **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. Análisis y comparación de las soluciones implementadas

## 3.1 Tiempos de ejecución

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 Cantidad de memoria utilizada

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

## 3.3 Complejidad de la implementación

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]](#doce)

## 4. Conclusiones

## 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. 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