Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 1

Distancia de Edición en Memoria Externa

Integrantes: Valeria Guidotti
Valeria Valdés
Maximiliano Vargas
Profesores: Gonzalo Navarro
Auxiliares: Dustin Cobas
Bernardo Subercaseaux
Ayudantes: Javier Morales
Matías Rojas
Github: https://git.io/Jendk

Fecha de entrega: 1 de octubre de 2019
Santiago, Chile

Índice de Contenidos

Lista de Figuras

Lista de Tablas

Lista de Códigos

1. Introducción

La distancia de edición entre dos strings $\mathcal{X} = x_1x_2x_3...x_n$ y $\mathcal{Y} = y_1y_2y_3...y_n$ es el número mínimo de *operaciones* requeridas para transformar una cadena en la otra. Las operaciones válidas son: **delete**(x_i), que elimina x_i de $\mathcal{X}$; **insert**(y_i) que inserta y_i en $\mathcal{Y}$; y **substitute**(x_i, y_j) que reemplaza x_i por y_j en $\mathcal{X}$.

En este informe se presentan los análisis, implementaciones y resultados experimentales de dos algoritmos utilizados para calcular la distancia de edición entre dos strings con longitud N en **memoria externa**.

Los algoritmos utilizan un *grafo de cuadrícula implícito* para $\mathcal{X}$ y $\mathcal{Y}$, de manera de aprovechar que la longitud del camino más corto desde la celda *(0,0)* a la *(N,N)* es la distancia de edición entre $\mathcal{X}$ y $\mathcal{Y}$.

Los algoritmos consideran que los strings $\mathcal{X}$ y $\mathcal{Y}$ son almacenados en N/B bloques en memoria externa. Uno de los algoritmos computa la matriz fila por fila, considerando lectura de cada bloque de la fila y computando los valores del bloque correspondiente en la nueva fila y escribiendo en memoria externa. El segundo algoritmo considera que se debe aprovechar la memoria principal para computar submatrices del grafo de cuadrícula implícito, leyendo bloques de strings en múltiplos de N/B.

2. Algoritmos

Los algoritmos trabajados se basan en el *grafo de cuadrícula implícito*, que para dos strings $\mathcal{X}$ y $\mathcal{Y}$ de largo N se define como:

- Una cuadrícula 2D de $(N + 1)$ x $(N + 1)$ nodos.
- Todo nodo tiene una arista a sus vecinos en el Sur(S), en el Este(E) y en el Sureste(SE).
- Las E-aristas y S-aristas tienen peso 1.
- Las SE-aristas $(i - 1, j - 1) \rightarrow (i, j)$ tiene peso 0 si $\mathcal{X}[i] = \mathcal{Y}[j]$, y peso 1 en otro caso.

Con esta definición, se puede usar la cuadrícula y el siguiente teorema para computar las distancias de edición.

Teorema. *La longitud del camino más corto desde la celda (0,0) a la (N,N) en el grafo de cuadrícula implícito para $\mathcal{X}$ y $\mathcal{Y}$ es la distancia de edición entre $\mathcal{X}$ y $\mathcal{Y}$.*

A continuación se describe en grandes rasgos los algoritmos implementados.

2.1. Algoritmo en RAM: Programación Dinámica

Para este algoritmo se considera calcular la cuadrilla de manera que los valores de cada fila se guarden en el espacio ocupado por los valores de la fila anterior, de esta manera no se requieren almacenar K^2 celdas en RAM para una matrix de KxK. Por lo tanto, en vez de trabajar con una matriz de $(N+1)$x$(N+1)$, se trabajará con dos filas de $(N+1)$, iterando por la cantidad de filas que posea la matriz.

El algoritmo descrito a continuación no utiliza programación dinámica, pero es fácilmente adaptable a las condiciones descritas en el párrafo anterior.

1. Construir una matriz de $(N+1)$x$(N+1)$ celdas, donde la casilla (i, j) contendrá la longitud del camino más corto desde (0,0) hasta (i, j).
2. Llenar la primera fila y la primera columna con los valores 0, 1, ..., n consecutivamente.
3. Calcular el valor para cada casilla secuencialmente, de izquierda a derecha, y de arriba hacia abajo. La casilla (i, j) es calculada como el mínimo valor entre:
   - El valor en $(i, j-1) + 1$
   - El valor en $(i-1, j) + 1$
   - El valor en $(i-1, j-1)$ si $\mathcal{X}[i] = \mathcal{Y}[j]$, o el valor en $(i-1, j-1) + 1$ si no.
4. Retornar el valor en la casilla (N, N).

2.2. Algoritmo en memoria externa 1: Algoritmo para RAM adaptado

Este algoritmo es el mismo que el descrito en la sección de Algoritmo en RAM, con la diferencia es que al computar la matriz fila por fila, se leerá cada fila bloque a bloque de la memoria externa, computando un bloque de la nueva fila y sobre escribiendolo en el bloque correspondiente de la fila anterior, reutilizando bloques previamente escritos.

2.3. Algoritmo en memoria externa 2: Particionando la cuadrilla

Este algoritmo aprovecha la capacidad M de la memoria principal, de manera de computar sub-cuadrillas del grafo de cuadrícula implícito. La idea es la siguiente:

1. Dividir la cuadrilla en subtablas con fronteras solapadas.
2. Procesar las subtablas de izquierda a derecha y de arriba hacia abajo. Por cada subtabla
   a) Leer de memoria externa los substrings y fronteras de entrada correspondientes.
   b) Computar los valores para la subtabla usando el algoritmo descrito en la sección de Algoritmo en RAM.
   c) Escribir las fronteras de salida a memoria externa.

3. Análisis Teórico

3.1. Algoritmo 1

Primero calculamos la memoria mínima necesaria para ejecutar este algoritmo, usando archivos de texto de un bloque de tamaño B cada uno. Primero se necesitan traer dos bloques a memoria para los dos strings, luego para calcular una fila usando la anterior se necesitan 4B para tener la fila anterior y 4B para computar la nueva fila, porque cada arreglo de enteros pesa 4B. Luego se necesitan 10B en memoria para ejecutar un paso del algoritmo.

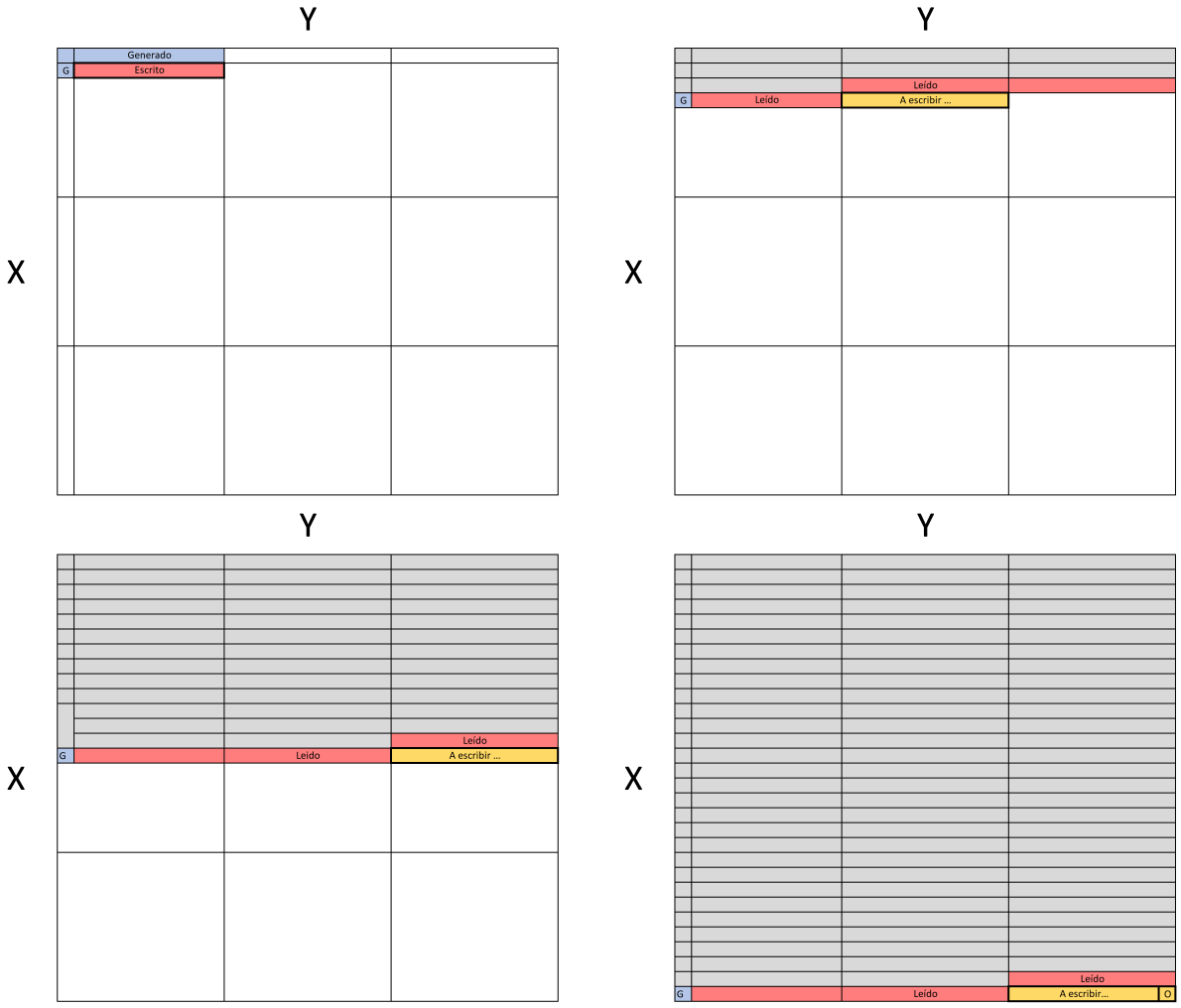

Figura 1: Visualización de iteraciones del algoritmo 1.

Ahora calculamos la cantidad de I/O: Un string X de tamaño N, se puede dividir en N/B bloques, para bloque del string X se computan las filas de la matriz comparando con el string Y, para un bloque del string X se tiene:

1. Leer bloque del string X $\implies$ 1
2. Leer Y $\implies \frac{N}{B}B = N$
3. Leer fila anterior $\implies \frac{N}{B}4B = 4N$
4. Escribir nueva fila $\implies \frac{N}{B}4B = 4N$

Lo que da un total de $(9N + 1)$ por bloque de X, entoces para todos los bloques de X se tiene un total de $\frac{N}{B}(9N + 1) \implies O(\frac{N^2}{B})$ I/O's.

3.2. Algoritmo 2

Primero se debe calcular la cantidad de memoria que se necesita para correr el algoritmo, al usar archivos de texto de tamaño B cada uno. Se traen f bloques para cada string, luego se trae la frontera superior y lateral izquierda, cada una ocupando $4f$ bloques cada frontera, entonces se tienen $8f$ bloques para las fronteras anteriores. Para las nuevas fronteras por el mismo argumento anterior se necesitan $8f$ bloques más. Entonces se necesitan $18f$ bloques. Se tiene memoria m y aproximamos la cantidad de bloques a $20f$ para variables que se mantienen en memoria principal. Despejando f:

$$20f = m \implies f = \frac{m}{20} \tag{1}$$

Así el algoritmo se va adaptando para usar la mayor cantidad de memoria posible

Para el análisis de I/O's se define K como la cantidad de bloques de un string en una submatriz. Por lo tanto, la cantidad de submatrices es:

$$\frac{N}{B}\frac{1}{K}\frac{N}{B}\frac{1}{K} \tag{2}$$

Para poder computar una submatriz es necesario que quepa en memoria:

$$K2 + K16 \leq \frac{M}{B} \implies K = O(\frac{M}{B}) \tag{3}$$

Se tiene que hacer esa cantidad de I/O's por cada submatriz, entonces para toda la matriz la cantidad de I/O's es:

$$\frac{N^2}{B^2K^2}K \implies O(\frac{N^2}{BM}) \tag{4}$$

Entonces se puede ver que el algoritmo 2 es más rápido que el algoritmo 1

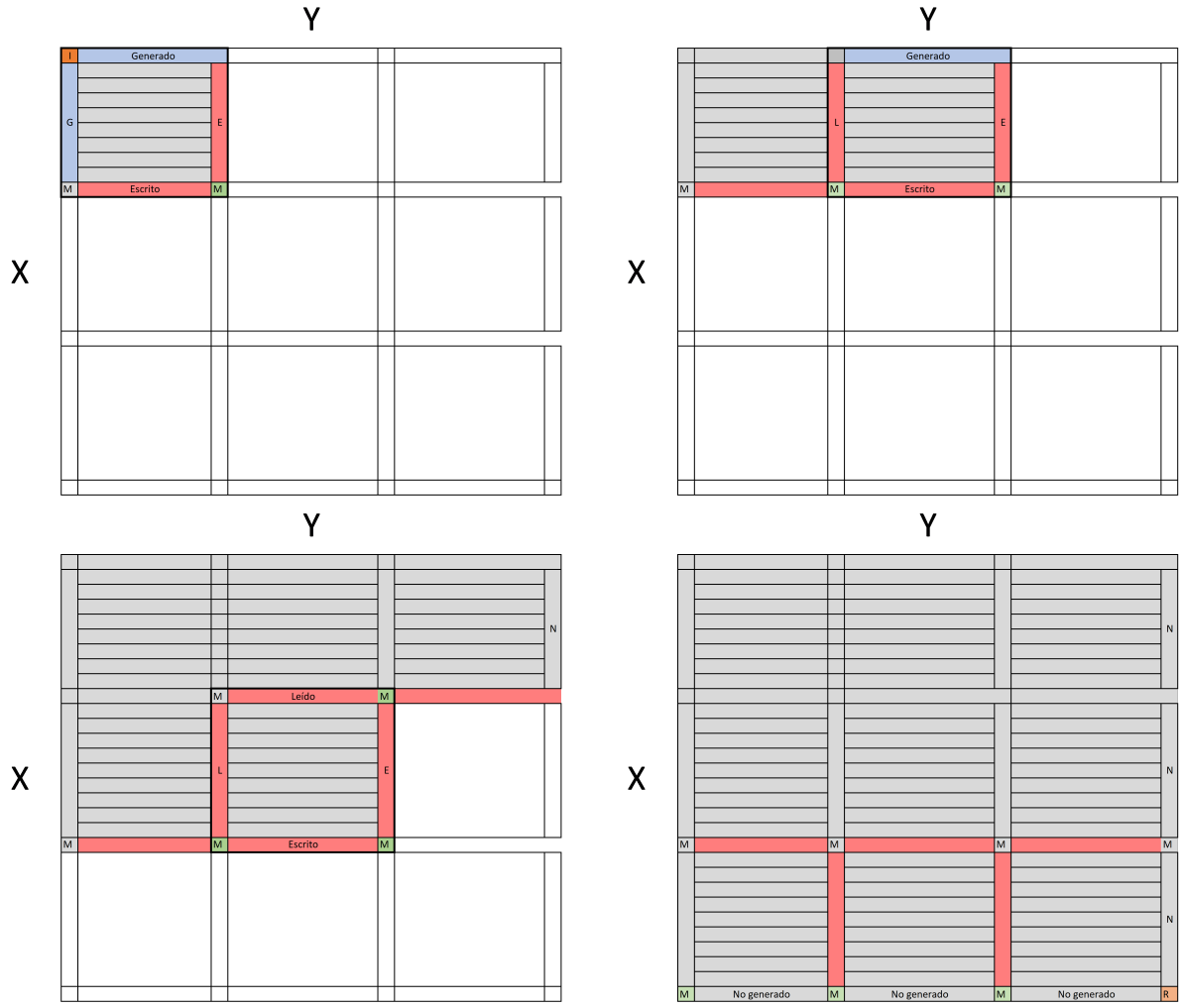

Figura 2: Visualización de iteraciones del algoritmo 2.

4. Implementación

4.1. Supuestos y definiciones

4.1.1. Valores, archivos

Se utilizarán 1024 bytes como tamaño de memoria **B**. Además, dado el tamaño **N** del input de X e Y en, se crearán tantos archivos como bloques necesarios, es decir, X e Y se dividirán en **N** archivos de tamaño **B** numerados convenientemente para su lectura/escritura.

4.1.2. Valor del factor f en segundo algoritmo

Notar que se puede describir a m en función del número bloques de un string que se traen a memoria en cada iteración. Además la cantidad de bloques depende de un factor f mecionado

anteriormente. Si se necesitan dos bloques para guardar variables locales, y 18 bloques para cada iteración de I/O, entonces $m = 18 * f + 2$. Sin embargo se usarán $m = 20 * f$ dado que $m \in 20, 40, 80$, todas son múltiplo de 20 (que facilitan la implementación). Al despejar f en ambas ecuaciones tenemos que:

$$f_{real} = \frac{m}{18} - \frac{1}{9} \quad f_{usado} = \frac{m}{20} \tag{5}$$

Pero hay que tener cuidado, ya que si por ejemplo se tienen $m = 24$ para ambos, f dará un valor $f \in [1, 2]$ y se necesitan enteros, por lo tanto se aplica la función *piso* o *floor*:

$$f_{real} = \lfloor \frac{m}{18} - \frac{1}{9} \rfloor \quad f_{usado} = \lfloor \frac{m}{20} \rfloor \tag{6}$$

Si se analizan, el error $f_{real} - f_{usado}$ queda que para un $m = 38$ la diferencia entre los factores es de 1, mientras que para $m = 218$ la diferencia es de 2, y para un $m = 398$ la diferencia es de 3. Sin embargo el error no es monótonamente creciente ya que por ejemplo, para $m = 40$ ambas funciones vuelven a valer lo mismo. Finalmente, para el rango de valores de m que se van a usar, f vale uno más sólo para $m = \{38, 39, 56, 57, 58, 59, 74, 76, 77, 78, 79\}$, por lo tanto para los valores de m a usar: 20,40,80 es indiferente usar f_{real} de f_{usado}.

4.2. Algoritmo para RAM modificado

Antes de ejecutar el algoritmo se inicializa la clase con los valores mostrados en el anexo A

Código 1: Pseudocódigo de algoritmo 1.

```
int SIZE_OF_INT = 4

int calculateDistance(){
    computeMatrix();
    return último elemento de la fila actual
}

void computeMatrix(){
    for(fila i 0, ..., N de X){
        if((i \% B) == 0){
            leer bloque i/B de x
        }
        computeRow(i)
    }
}

void computeRow(int row){
    for(bloque j 0, .., N/B de Y){
        leer bloque j de Y
        computeBlockOfFile(j, row)
    }
}
```

computeBlockOfFile computa el bloque del archivo y escribe la nueva fila, *computeBlockOfMAtrix* ejecuta el algoritmo, actualiza filas y variables y llama a *actualizePreviousRow* que lee la fila anterior, los códigos fuente de estas funciones se pueden ver en el anexo B.

4.2.1. Análisis

Como el algoritmo se implementó como es originalmente la cantidad de I/O's es la mencionanda en el análisis teórico $O(\frac{N^2}{B})$. La complejidad computacional es $O(N^2)$.

El uso de memoria secundaria corresponde a $\frac{2N}{B}$ para los strings y $\frac{4N}{B}$ para las filas, entonces el total de memoria secundaria usada es $\frac{6N}{B}$

4.3. Algoritmo particionando cuadrilla

Antes de ejecutar el algoritmo se inicializa la clase con los valores mostrados en el anexo C

Código 2: Pseudocódigo de algoritmo 2.

```
//Variables importantes de la clase
int cantSub
int[] diagValues

int calcAllDist(){
for(submatriz i perteneciente a las filas de X){
    x = leer bloques correspondientes a la submatriz i
    for(submatriz j perteneciente a las filas de Y){
        y = leer bloques correspondientes a la submatriz j
        //Obtener frontera lateral
        if(j == primera columna de submatrices){
            Frontera lateral anterior = Generar frontera lateral de X
        }
        else{
            Frontera lateral anterior = Leer frontera lateral correspondiente a
    submatriz j
        }

        //Obtener frontera superior
        if(i == primera fila de submatrices){
        Frontera superior anterior = Generar frontera superior
        }
        else{
        Frontera superior anterior = Leer frontera superior correspondiente a submatriz j
        }
        calcDist(j,i)
    }
}

    Return último valor de frontera superior nueva
}
```

```
31
32 void calcDist(j, i){
33     int largo <- B*f
34     for (fila 0,...,largo-1){
35         computarFilaB(fila, j)
36         diagValues[j] = Último valor de frontera lateral anterior
37         if(j es la última submatriz){
38             Escribir solo la nueva frontera superior
39         }
40         else{
41             if(i es la última fila de submatrices){
42                 Escribir solo la nueva frontera lateral
43             }
44             else{
45                 Escribir nueva frontera lateral
46                 Escribir nueva frontera superior
47             }
48         }
49     }
50 }
```

La función $computarFilaB(int fila, int j)$ es la encargada de realizar el algoritmo en RAM, además de guardar las nuevas fronteras en el objeto.

4.3.1. Análisis

A pesar de que se optimizó el algoritmo para que no escriba las fronteras de las últimas columnas y no escriba las fronteras de la última fila (notar en Figura 2), la cantidad de I/O's sigue siendo la mencionada anteriormente. La complejidad computacional es $O(N^2)$

El uso de memoria secundaria corresponde a $\frac{2N}{B}$ para los strings, $(\frac{m}{5} - 1)B$ para las fronteras laterales y $\frac{m}{5}B$ para las fronteras superiores.

4.4. Dificultades

4.4.1. El NO uso de *multithread*

Puede parecer razonable crear múltiples threads para ejecutar los experimentos de los algoritmos en paralelo y asegurar un uso de 100 % de CPU todo el tiempo. Sin embargo, esto es una mala idea porque si bien la CPU estará ocupada, la asignación de recursos de CPU a cada thread será irregular, debido a que esto lo controla el ***system scheduler*** dependiente de cada OS, lo que provocaría una medición errónea de tiempo de ejecución de cada algoritmo.

Por otro lado, la arquitectura de un disco duro magnético consta de un número límitado de agujas para la lectura/escritura de bytes, a veces incluso sólo una[2]. Por lo tanto, la escritura/-lectura en "paralelo" en disco es una ficción por límites físicos. Quizás es posible usar más de un disco, corriendo al menos dos threads en directorios de discos diferentes pero incluso en este caso no podemos predecir al system scheduler, siendo mejor usar dos computadores.

4.4.2. Alternativas de lectura/escritura de archivos

Durante la realización de la tarea se encontraron dos posibilidades de lectura de archivos. La primera forma era leer byte a byte un archivo hasta llenar un buffer en memoria generando tantos I/O como bytes leídos. La segunda forma, era leer un arreglo de bytes en un sólo I/O, como se esperaría al leer un bloque. Ambas formas se describen como sigue:

Código 3: Lectura secuencial de bytes en buffer.

```
1    while( dataInputStream.available() > 0) { // Leer hasta el final
2        X[k] = dataInputStream.readByte(); // Byte a byte
3        k++; // Avanzar
4    }
```

Código 4: Lectura de arreglo de bytes.

```
1    byte[] temp = new byte[B]; // Arreglo temporal
2    dataInputStream.read(temp); // Leer un bloque directamente
3    System.arraycopy(temp, 0, X, i*B, B); // Copiar a arerglo global
```

Ambas formas se contaron como un I/O en el código, pero se notó que la primera forma hacía demorar a el primer algoritmo mucho más. Por lo tanto se decidió usar la segunda forma para la recopilación y análisis de resultados.

5. Ejecución

5.1. Instrucciones y descripción

La clase *Experiments* realiza 15 experimentos para cada N para el primer algoritmo y 15 experimentos para cada combinación N, M para el algoritmo 2.

En el caso del algoritmo 1, solo se realizan experimentos para los distintos N sin considerar los valores de M, pues el algoritmo toma la misma cantidad de accesos a disco para un mismo tamaño de string, independiente de la cantidad de memoria principal disponible.

Para ejecutar los experimentos, en el directorio *src*, deben existir los directorios *out/results* y *out/files*, en el primero de estos últimos directorio quedan guardados los resultados de los experimentos y la segunda se usa para los archivos que los algoritmos escriben.

Para ejecutar los experimentos se debe ingresar los comandos:

javac Experiments.java

java Experiments

5.2. Limitaciones de *Java*

La principal limitante de usar **Java** como lenguaje, tiene que ver con los límites de memoria RAM mínima y máxima que se pueden asignar a una rutina en la JVM. Para la memoria mínima se usa el comando **-Xms** y tiene un valor mínimo de 1Mb, por otro lado, para la memoría máxima se usa **-Xmx** y tiene un mínimo de 2Mb [1]. Por lo tanto es imposible dar 20K de memoria al programa (el equivalente a 20 bloques de 1Kb). A causa de esto, la forma de medir la eficiencia de cada algoritmo fue mediante I/O, tiempo y otros parámetros.

6. Resultados

Los resultados de los 15 experimentos descritos en la sección anterior fueron obtenidos luego de su ejecución en un sistema con las siguientes características:

- Procesador: Intel Celeron N3350 @ 1.10GHz
- Memoria RAM: LPDDR4 4GB
- Almacenamiento: SSD SATA de 256 GB
- Sistema Operativo: Ubuntu 18.04 LTS

El lenguaje de programación utilizado fue Java 8, y OpenJDK 8 como SDK. Como IDE se utiliza IntelliJ IDEA IU 2018.3. El tiempo total de ejecución de los experimentos fue de 6 horas aproximadamente.

6.1. Algoritmo para RAM modificado

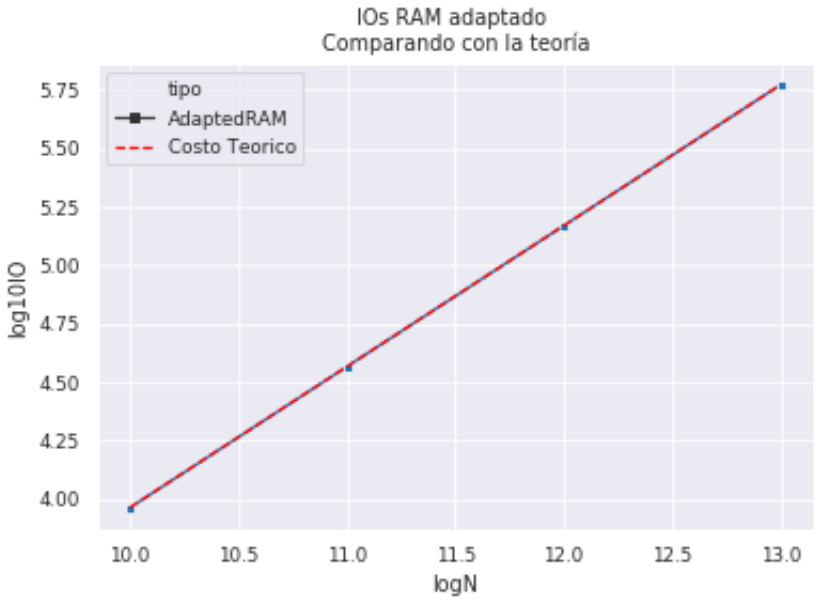

Figura 3: Comparación de I/O's teórica v/s experimental en función de N para algoritmo RAM adaptado en escala logarítmica.

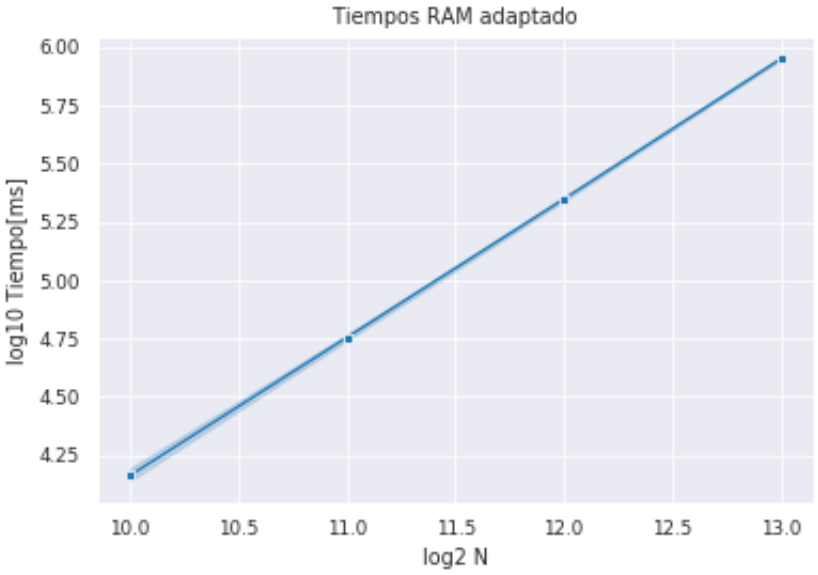

Figura 4: Tiempo de ejecución en función de N para algoritmo RAM adaptado en escala logarítmica.

La tabla con los resultados de desviación estándar, promedio y mediana de los tiempos se encuentra en la tabla 1

6.2. Algoritmo particionando cuadrilla

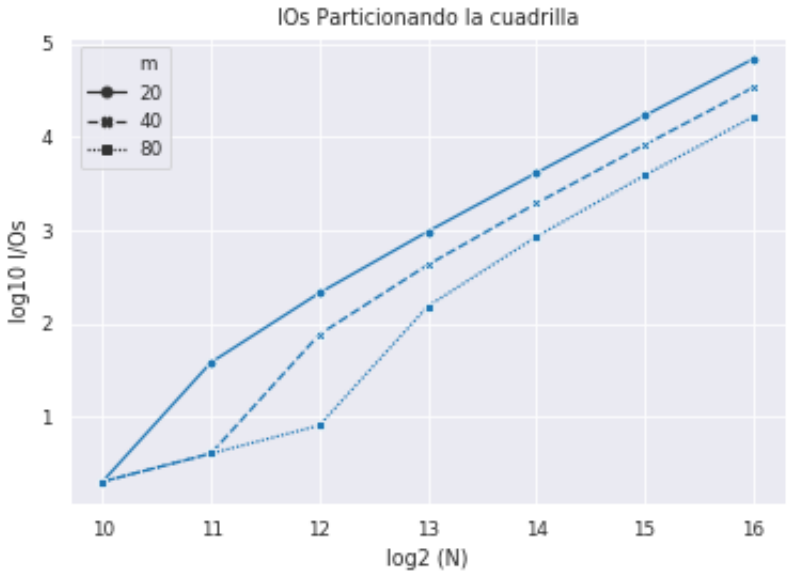

Figura 5: Cantidad de I/O's experimental en función de N para distintos valores de m en escala logarítmica para algoritmo particionado en cuadrilla.

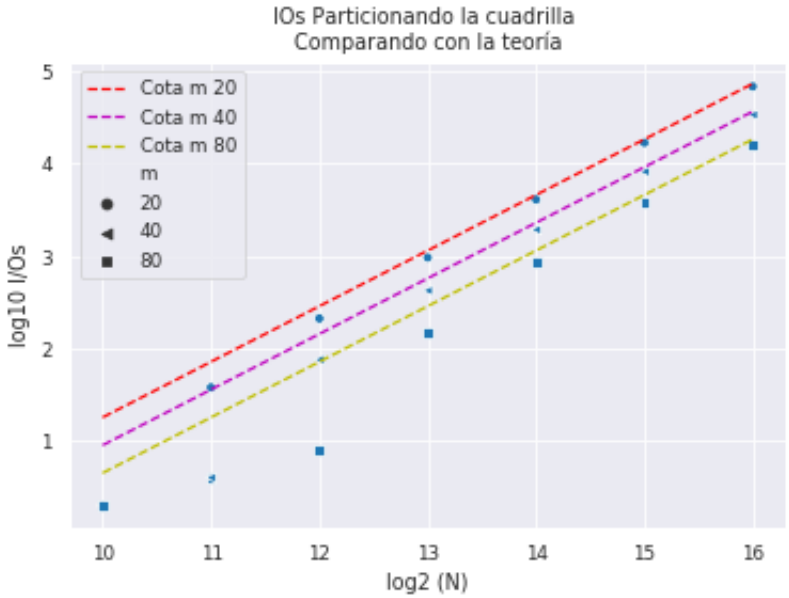

Figura 6: Cantidad de I/O's teórico v/s experimental en función de N para distintos valores de m en escala logarítmica para algoritmo particionado en cuadrilla.

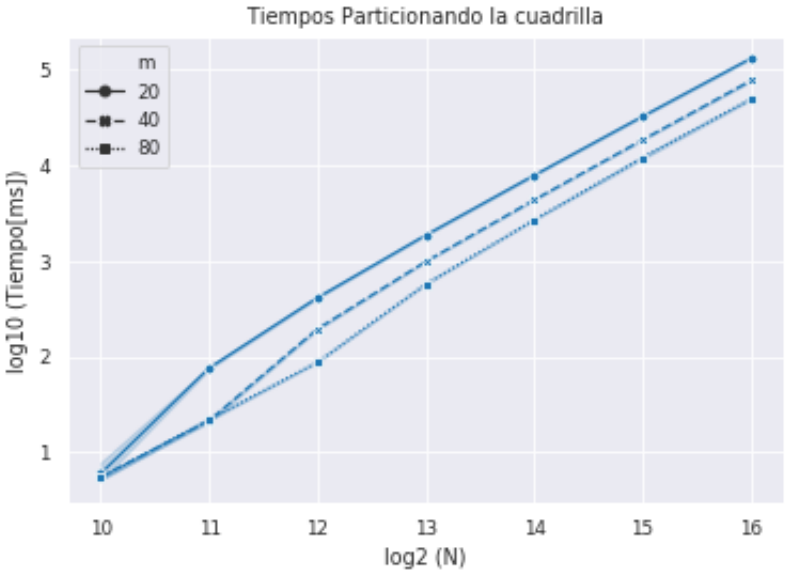

Figura 7: Tiempo de ejecución en función de N para distintos valores de m en escala logarítmica para algoritmo particionado en cuadrilla.

La tabla con los resultados de desviación estándar, promedio y mediana de los tiempos se encuentra en la tabla 2

6.3. Comparación de ambos algoritmos

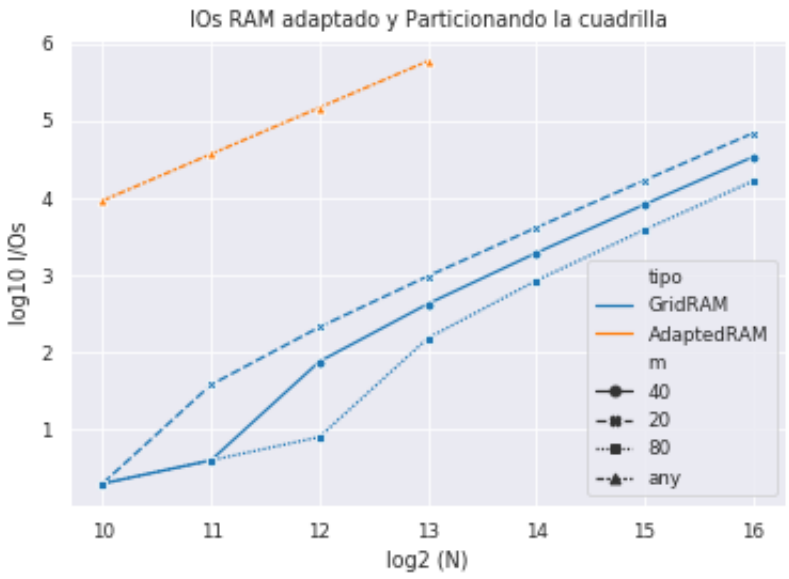

Figura 8: Comparación de I/O's en función de N para algoritmo RAM adaptado y algoritmo particionado en cuadrilla para distintos valores de m en escala logarítmica.

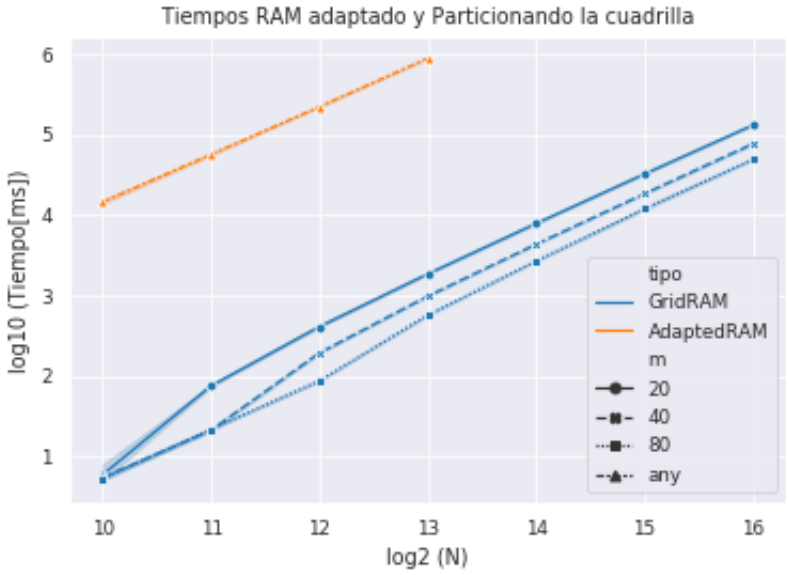

Figura 9: Comparación de tiempo de ejecución en función de N para algoritmo RAM adaptado y algoritmo particionado en cuadrilla para distintos valores de m en escala logarítmica.

7. Análisis y Discusión

De la Figura 3 se concluye que el costo teórico de I/O es asintótico con los puntos obtenidos de los resultados experimentales, lo cual valida la cota calculada y el análisis previo realizado. Por otro lado, es esperable que a medida que crezca el input, crezca también la cantidad de I/O necesarios.

La desviación estándar graficada en la Figura 4, que cuesta apreciar, muestra que los experimentos realizados fueron precisos, es decir, que el error de las mediciones de tiempo es bajo para cada N. Por otro lado muestra que evidentemente el tiempo demorado aumenta al crecer N.

Además, de la Tabla 1 (en sección tablas del anexo) puede concluirse que el error asociado a las mediciones aumenta con el tamaño del N. Esto puede deberse a pequeñas inestabilidades al ejecutar el algoritmo como el pequeño error acumulado en milisegundos que tarde una lectura/escritura.

De la Figura 5, se puede concluir que el crecimiento en I/O es similar para todos los valores de m. Sin embargo, es claro que a medida que la memoria crece, la cantidad de I/O decrece. Esto se debe a que en el algoritmo de particionado de cuadrilla, la memoria se llena lo más posible para ejecutar cada iteración del algoritmo.

De la Figura 6, se puede concluir que la cota teórica de I/O calculada es cota para los valores de los resultados experimentales, para todos los valores de m. Además, al aumentar N es posible concluir que los valores de los resultados experimentales se acercan a los valores teóricos calculados.

Para $log_2 N = 10$, es posible apreciar que para las tres m el algoritmo de particionado de cuadrilla tiene la misma cantidad de I/Os. Esto se debe a que para todas los tamaños de memoria, los string X e Y de este tamaño caben por completo en memoria, por lo que en los tres casos realiza la misma cantidad de accesos a disco. Para $log_2 N = 11$, X e Y caben por completo en memoria cuando $m = 40, 80$ pero no $m = 20$, por lo que los dos primeros casos tienen la misma cantidad de I/Os. para $log_2 N = 12$, el input sólo cabe completo en RAM para $m = 80$. Es posible observar que para los primeros N la cantidad de I/Os no se ajusta a la cota, esto se debe a la optimización de lecturas y escrituras realizada descrita en la seccion 4.3, que no escribe las columnas de la derecha de la matriz ni las filas inferiores. Para los siguientes valores de N, la cantidad de I/Os se comporta bajo la cota calculada.

En la Figura 7 es posible apreciar en medida de tiempo lo recién descrito. En el primer valor de N todos los algoritmos demoran lo mismo porque todo el input cabe en memoria, y el análisis a partir de aquí es extensible del análisis del párrafo anterior.

De la Figura 8 y 9, es posible concluir que la cantidad de I/O y tiempo requerida por el algoritmo de RAM adaptado es en gran medida mayor al algoritmo de particionado de cuadrilla. Para distintos valores de N pueden tener hasta 3 órdenes de diferencia en tiempo e I/O. Esto se debe a que el primer algoritmo realiza muchos más I/O que el segundo, tal como se describe en el análisis teórico.

8. Conclusión

Luego de una revisión de resultados se puede concluir que el orden de la cantidad de I/O's experimentales coincide con el teórico para el algoritmo de RAM adaptado y se acerca para el algoritmo particionado en cuadrilla, esto porque el cálculo teórico fue de cota superior. También se puede observar que el algoritmo particionado en cuadrilla para valores pequeños de N, como en la implementación se optimizó, realiza menos I/O's que las esperadas por la teoría.

Se puede ver que los gráficos de I/O's se parecen a los de tiempo de ejecución, esto es debido a que el tiempo de ejecución se debe relacionar linealmente con la cantidad de accesos a disco, ya que estas son la acciones que más tiempo toman en los algoritmos de memoria externa.

Finalmente, en la comparación de ambos algoritmos se puede concluir que el segundo algoritmo es más rápido que el primero, esto porque adapta la computación aprovechando de mejor forma la memoria principal disponible, usándola lo más posible.

Referencias

[1] Oracle. “Java Documentation, -Xmssize”. [En línea]. Disponible en: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html

[2] hdd-tool.com “hard disk and hard drive physical components”. [En línea]. Disponible en: https://www.hdd-tool.com/hdd-basic/hard-disk-and-hard-drive-physical-components.html

Anexos

A. Inicializador algoritmo para RAM modificado

Código 5: Inicializador algoritmo para RAM modificado.

```
public AdaptedRAM(int N){
        this.N = N;
        this.I = 0;
        this.O = 0;

        this.westValue = 1;
        this.northWestValue = 0;

        this.X = new byte[B];
        this.Y = new byte[B];

        this.previousRow = new int[numberOfInts];
        this.actualRow = new int[numberOfInts];
        for(int i = 0; i < numberOfInts; i++){
            this.previousRow[i] = i + 1;
        }
    }
```

B. Funciones de RAM modificado

Código 6: Funciones de RAM modificado.

```
public void computeBlockOfFile(int stringBlock, int row){
        for(int i = 0; i < SIZE_OF_INT; i++){
            computeBlockOfMatrix(i, stringBlock, row);
            if(!(stringBlock == N/B - 1 && row== N - 1)) writeActualRowToFile(i,
    stringBlock);
        }
}
public void computeBlockOfMatrix(int matrixSubBlock, int stringBlock, int row){
        int nwVal, nVal, wVal;

        actualizePreviousRow(matrixSubBlock, stringBlock, row);

        wVal = westValue + 1;
        nVal = previousRow[0] + 1;
        nwVal = (Y[numberOfInts * matrixSubBlock] == X[row % B]) ?
                northWestValue : northWestValue + 1;
        actualRow[0] = Math.min(nwVal, Math.min(nVal, wVal));

        for(int i = 1; i < numberOfInts; i++){
```

```
            wVal = actualRow[i-1] + 1;
            nVal = previousRow[i] + 1;
            nwVal = (Y[numberOfInts * matrixSubBlock + i] == X[row % B]) ?
                    previousRow[i - 1] : previousRow[i - 1] + 1;
            actualRow[i] = Math.min(nwVal, Math.min(nVal, wVal));
        }

        actualizeNWValues(matrixSubBlock, stringBlock, row);
}

public void actualizePreviousRow(int matrixSubBlock, int stringBlock, int row){
        if(row == 0){
            if(!(matrixSubBlock == 0 && stringBlock ==0)){
                for(int i = 0; i < numberOfInts; i++){
                    previousRow[i] = previousRow[i] + numberOfInts;
                }
            }
        } else {
            readIntoPreviousRow(matrixSubBlock, stringBlock);
        }
}
```

C. Inicializador algoritmo particionado cuadrilla

Código 7: Inicializador algoritmo particionado cuadrilla.

```
public RAMConFronteras(String dir_x, String dir_y, String dir_output, int m, int N){
        this.I = 0;
        this.O = 0;
        this.N = N;
        this.dir_x = dir_x;
        this.dir_y = dir_y;
        this.dir_output = dir_output;
        this.f = N/B < m/20 ? N/B : m/20;
        this.newFrontierColumn = new int[f*B];
        this.newFrontierRow = new int[f*B];
        this.previousFrontierColumn = new int[f*B];
        this.previousFrontierRow = new int[f*B];
        this.cantSub = (int) Math.ceil(N/(B*f));
        this.diagValues = new int[cantSub];
        for(int i = 0; i < cantSub; i++){
            diagValues[i] = B * f * i;
        }
        this.X = new byte[B*f];
        this.Y = new byte[B*f];
}
```

D. Tablas

Tabla 1: Promedio, desviación estándar y mediana de tiempos (en milisegundos) de los experimentos del algoritmo de RAM adaptado para distintos tamaños de string (N).

N	mean	std	median
1024	14710.07	2302.96	13886
2048	56986.87	3815.78	55817
4096	223083.80	8931.54	222492
8192	888865.20	14797.80	896497

Tabla 2: Promedio, desviación estándar y mediana de tiempos (en milisegundos) de los experimentos del algoritmo de "particionando la cuadrilla"para distintos tamaños de string (N) y de bloques en memoria (m).

N	m	mean	std	median
1024	20	7.60	9.25	5
	40	5.53	0.52	6
	80	5.4	0.51	5
2048	20	76.27	7.19	74
	40	21.47	2.42	21
	80	22.07	1.28	22
4096	20	414.53	17.49	407
	40	195.00	9.72	190
	80	87.73	1.79	87
8192	20	1869.80	43.72	1872
	40	988.47	15.36	988
	80	576.33	18.75	570
16384	20	7903.60	163.48	7888
	40	4363.20	57.58	4367
	80	2705.73	43.67	2713
32768	20	32590.47	460.49	32668
	40	18517.13	277.33	18587
	80	12110.73	1491.38	11724
65536	20	131809.13	2147.48	132691
	40	77329.07	4495.13	76679
	80	49620.07	2386.81	49106