Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
CC4102 - Diseño y Análisis de Algoritmos

Diseño y Análisis de Algoritmos

Intersección de Segmentos Ortogonales

Tarea 1

Profesor:
Pablo Barceló

Auxiliar:
Ariel Cáceres

Ayudantes:
Claudio Torres
Jaime Salas

Integrantes:
Juan Andrés Moreno
Tomás Perry

Fecha:
25 de mayo de 2017

Índice general

Capítulo 1

Resumen

En este informe se trabaja el problema de “Intersección de segmentos ortogonales”, el cual es utilizado , mayormente , en computación geométrica. Debido a los grandes volúmenes de datos que se manejan en esta area resulta de interés la implementación eficiente de un algoritmo que opere en memoria secundaria. Se entiende por eficiente un algoritmo que optimice la cantidad de accesos a disco.
Primero se explicará el problema en detalle y la metodología ocupada para resolverlo, además de los factores a considerar para obtener una solución óptima. Luego se presentan los resultados. Se analizan estos tomando en cuenta lo que se desea optimizar, intentando identificar los aciertos y errores.
Por ultimo se presentan los resultados obtenidos para ambos algoritmos implementados y para todas las entradas pedidas por enunciado. Con estos, se analiza y se concluye sobre si se logro el orden de accesos a disco y tiempo de ejecución esperados.

Capítulo 2

Introducción

El problema de "Intersección de segmentos ortogonales" consiste en, a partir de un conjunto de segmentos verticales y horizontales, identificar todas las intersecciones entre estos, evidentemente siendo entre segmentos ortogonales. En particular, se trabaja con conjuntos suficientemente grandes para no ser procesables usando únicamente memoria principal, significando que se estudian por subconjuntos extraídos de memoria secundaria.
Para resolver el problema en esta tarea se propone implementar el algoritmo *Distribution Sweep*. Este requiere, entre otras cosas, generar listas de segmentos ordenadas por alguna coordenada. Dado que se trabajan con entradas de tamaño mayor a memoria principal, se tiene entonces que es necesaria la implementación de algún algoritmo de ordenamiento en memoria secundaria. Para esta tarea se decidió implementar Merge Sort.

2.1. Distribution Sweep

Es este el método que finalmente identifica las intersecciones, ocupando *slabs* para separar los segmentos en regiones y, recorriendo un *slab*, ver qué segmentos verticales intersectan a los segmentos horizontales. Este se explicará en detalle en la metodología.

Capítulo 3

Metodología

3.1. Datos

3.1.1. Input

Para estudiar el algoritmo, es necesario entregar una lista de segmentos, horizontales y verticales, definidos por la coordenada de inicio y de término. Para tener entradas suficientemente grandes para justificar la implementación de algoritmos que operen en memoria secundaria se generaron archivos de prueba con 2^i segmentos, donde $i \in \{9, 10, ..., 21\}$.
Era necesario controlar una cantidad razonable de intersecciones entre segmentos. Para esto se decidió acotar los valores posibles que puede tomar una coordenada por $\pm 2^{\frac{i}{2}}$, con $i \in \{9, 10, ..., 21\}$. Para el caso de distribuciones uniformes esto fue directo. Por otro lado, para el caso de las distribuciones normales se utilizo $\mu = 0$ y ademas se ocupo un valor de σ tal que el 90 % de los datos se encontraran en el rango $\pm 2^{\frac{i}{2}}$. Considerando que el 90 % de los datos se encuentran a $\pm 1{,}65\sigma$ del centro de la distribución, se tiene que con $\sigma = \frac{2^{\frac{i}{2}}}{1{,}65}$ se logra lo querido. Esto se implemento utilizando la librería *random* en Python. Se generaron listas con una distribución normal para la coordenada x, y repitiendo para distribución uniforme.
Además, se varió la proporción de segmentos verticales y horizontales, tal que

$$\frac{\#_{segmentos\_horizontales}}{\#_{segmentos\_verticales}} = \frac{\alpha}{(1-\alpha)} \tag{3.1}$$

con $\alpha \in \{0{,}25, 0{,}5, 0{,}75\}$

3.1.2. Parámetros

Luego de generar los inputs de prueba (con script de Python adjunto a esta tarea) se comprobó que el tamaño de estos no era el necesario para justificar los accesos a memoria secundaria.
Definimos ciertos valores para regular los accesos a disco y verificar la eficiencia de nuestro algoritmo.

1. B: Tamaño del bloque de información que se trae por acceso a disco, en *bytes*.
2. M: Tamaño de memoria secundaria a ocupar, en *bytes*.

Ademas se cuenta con el parámetro:

1. N: Tamaño del input.

Con el fin de optimizar la forma en que se accede, se utilizan múltiplos de 4 para el *buffer*, asegurando así que no hayan bloques incompletos de información.

3.2. Algoritmos

3.2.1. Sort

Fue necesario diseñar un método de ordenamiento que funcionara en memoria secundaria, pudiendo elegir entre *MergeSort* y *DistributionSort*, optando por *MergeSort*, por la familiaridad con el algoritmo. Al igual que las particiones que se realizan en el *MergeSort* tradicional, en este algoritmo enfocado

en la optimización de accesos a disco, se utilizan $\Theta(m/n)$ "*runs*", siendo cada uno una dirección a un archivo en memoria secundaria. Dado que cada uno de estos es de tamaño $\mathcal{O}(m)$ se pueden ordenar en memoria principal por separado.
Luego, se realiza el *merge* entre $\Theta(m)$ *runs* de igual tamaño y poniéndolos nuevamente en memoria secundaria. Evidentemente los *runs* dejarán de caber en memoria principal, por lo que no se traen completos, sino se ocupa un *buffer* de entrada para cada uno y un *buffer* de salida para el *run* obtenido, todos los anteriores de tamaño $\mathcal{O}(B)$.

3.2.2. Distribution Sweep

Como dicho antes, es este algoritmo el que identifica las intersecciones. Se identifican los siguientes pasos.

1. Generar una lista ordenada según X de los segmentos, la llamaremos *Lista X*.
2. Generar una lista ordenada según Y de los segmentos, la llamaremos *Lista Y*.
3. Dividir en k *slabs* según la *Lista X*, con una lista para cada uno.
4. Recorriendo la *Lista Y*, iterando en el valor de la coordenada Y:
   - Agrego los segmentos verticales a los *slabs* a los que pertenezcan, ocupando la lista de cada *slab*, la llamaremos l. Una vez el valor en que se va de la iteración es mayor al segmento, se remueve de l.
   - Si es un segmento horizontal, se recorren las listas de los *slabs* a los que el segmento pertenece, sin considerar el *slab* en el que inicia y en el que termina, y se registran todos los segmentos verticales que estan en las listas, guardando las intersecciones.
   - Se repite el algoritmo desde 3 en cada *slab* hasta que este quepa en memoria principal.

Se eligió un B=4000 y un M=40000.

Capítulo 4

Resultados

Debido a un mal estudio del tiempo y el trabajo necesario, solamente se trabajo el *MergeSort* con acceso a discos, quedando pendiente el algoritmo de *Distribution Sweep*.
Se realizó el ordenamiento para los distintos tamaños de listas de segmentos aleatoriamente generados, variando las proporciones, y se midió el número de accesos a discos, comparando entre distribuciones y proporciones entre segmentos horizontales y verticales y una comparación entre todos. Cabe destacar que cada vez que se rellenaba un buffer se hacia un acceso a disco. Por otro lado, lo mismo ocurre al momento de escribir un buffer hacia su archivo correspondiente.

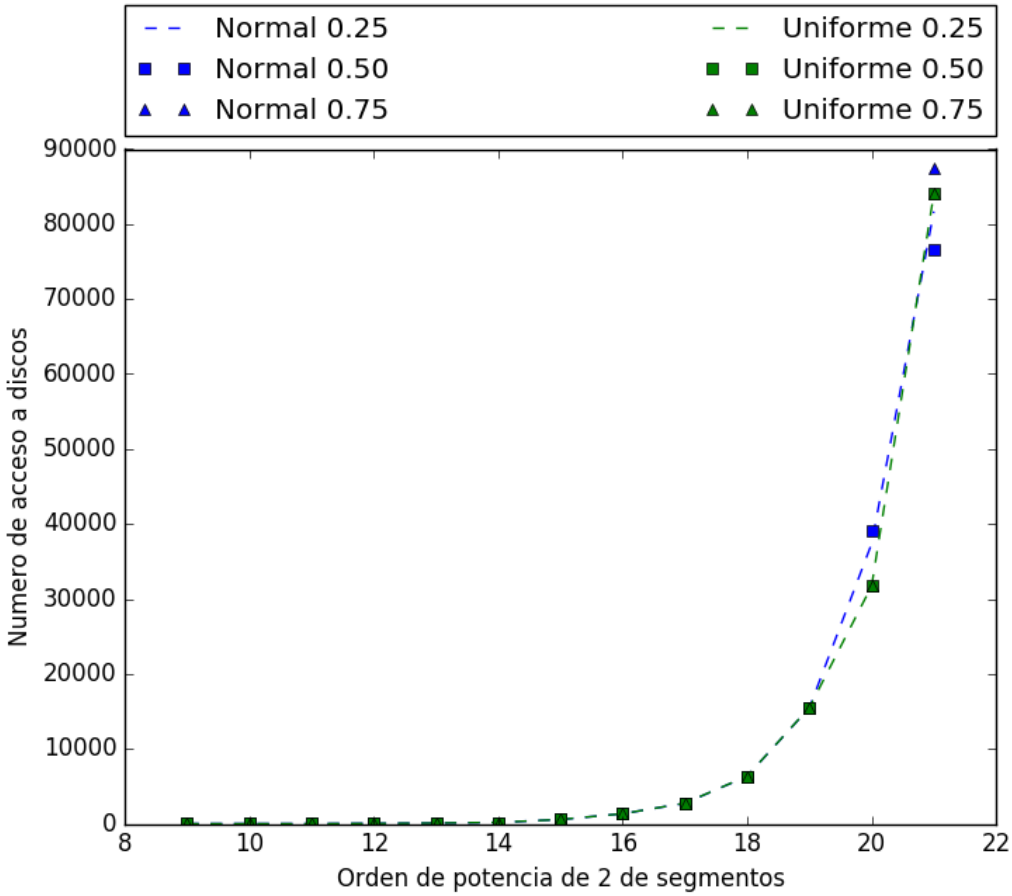
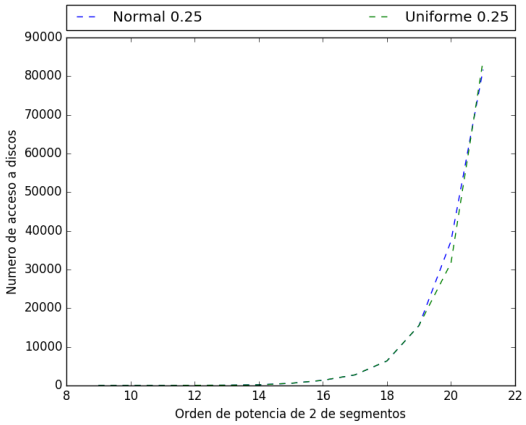
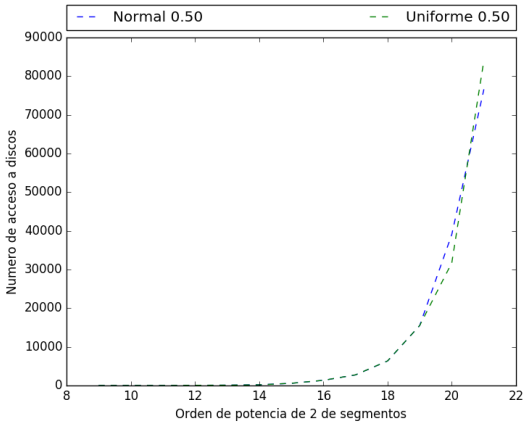


Figura 4.1: Comparación de todos los *mergesorts*.

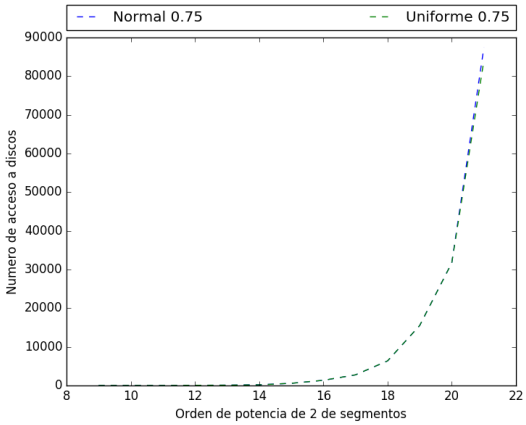
A partir de la Figura 4.1, se observan comportamientos similares entre el ordenamiento de listas con distribución normal como uniforme, teniendo un crecimiento similar, pero se observa que el acceso a discos es igual para todas las distribuciones uniformes de igual tamaño, sin importar la proporción entre segmentos verticales y horizontales, mientras que el caso de la distribución normal se observa que una mayor cantidad de segmentos verticales es mejor.



(a) Comparación para $\alpha = 0{,}25$



(b) Comparación para $\alpha = 0{,}50$



(c) Comparación para $\alpha = 0{,}75$

Figura 4.2: Comparaciones entre distribuciones, según su proporción entre segmentos horizontales y verticales.

A partir de la Figura 4.2, se puede observar que el ordenamiento de una distribución es superior a la otra dependiendo de la proporción entre segmentos verticales y horizontales, creciendo el número de accesos para las distribuciones normales y manteniéndose similar para las distribuciones uniformes. Por simplicidad, no se incluyen los resultados obtenidos al ordenar según la coordenada y, pues estos eran iguales.

Capítulo 5

Análisis

A partir de los datos obtenidos[1], se calcula que los accesos a disco son $\Theta(nlog_m(n))$, con $n = \frac{N}{B}$, $m = \frac{M}{B}$, con una constante cercana a 60, teniendo una constante alta, pero cumpliendo con el orden teórico.
La constante podría deberse a una mala elección de tamaño de bloque y/o memoria secundaria a ocupar, cambiando la base del logaritmo.
La diferencia obtenida entre ordenar una distribución normal y una uniforme podría deber a optimizaciones en el *Branch Predictor* del procesador, siendo mas eficiente en distribuciones uniformes.

[1] Adjuntos en el archivo 'results.csv'

Capítulo 6

Conclusiones

A partir de la experiencia, es necesario destacar el *cuello de botella* que se genera al requerir accesos a disco, ralentizando considerablemente la velocidad de cómputo y disminuyendo el uso útil de la CPU. Se señala el impacto que tiene la distribución sobre el ordenamiento, pudiendo deberse a optimizaciones que se hacen a nivel de la arquitectura del computador.
Se rescata la influencia de la proporción entre segmentos sobre el número de accesos a disco, pudiendo observar un beneficio en tener una proporción equitativa.

Capítulo 7

Anexos

7.1. Códigos

7.1.1. Generación de Datos

```
import numpy as np
import random
import pandas as pd
import math

alpha = [0.25, 0.5, 0.75]
N = list(range(9,22))
distributions = ['uniform', 'normal']
file_names = []

def generate_files():
    for n in N:
        for a in alpha:
            for d in distributions:
                f_name = 'input-2-{}-{}-{}.csv'.format(str(n),d,a)
                segments = generate_segments(n, d, a)
                df = pd.DataFrame(segments)
                df.to_csv(f_name, index=False, header=False)
                print("Finished generating input of size 2^{}, alpha = {} and distribution = {}\n".format(n,↩
                    a,d))

def generate_segments(N, distribution, alpha):
    vs_qty = int((2**N)*(1-alpha))
    hs_qty = int((2**N)*alpha)

    vs_ctr = 0
    hs_ctr = 0

    ##[U+FFFD]Usingsets make checking if a segment is in the generated set faster.
    segments = set()
    bound = math.floor(2**(N/2))
    ## ------------------------------------------------------------------------
    ## Vertical Segments generation:
    while True:
        ## ------------------------------------------------------------------------
        ## ------------------------------------------------------------------------
        ## First we generate x coordinates with its corresponding distribution:
        if(distribution == 'uniform'):
            x_1 = x_2 = math.floor(np.random.uniform(-bound, bound))
        elif(distribution == 'normal'):
            ## 1.0 : The idea here is to have 95% of the points between -2^(N/2) and 2^(N/2) .
            ##       Still not sure if it will behave well, we could need to change this.
            sigma = bound / 1.65
            x_1 = x_2 = math.floor(np.random.normal(0, sigma))

        ## ------------------------------------------------------------------------
        ## ------------------------------------------------------------------------

        ##[U+FFFD]Nowe generate the y coordinates:
        ## 1.1 : Not sure if we need to check that y_1 != y_2, could this really not happen?
        y_1 = math.floor(np.random.uniform(-bound, bound))
        y_2 = math.floor(np.random.uniform(-bound, bound))
        while(y_1 == y_2):
            y_1 = math.floor(np.random.uniform(-bound, bound))
            y_2 = math.floor(np.random.uniform(-bound, bound))

        ## ------------------------------------------------------------------------

        if((x_1, y_1, x_2, y_2) not in segments):
            segments.add((x_1, y_1, x_2, y_2))
            vs_ctr += 1
        if(vs_ctr == vs_qty):
            break
    ## ------------------------------------------------------------------------
    ## Horizontal Segments generation:
    while True:

        ## 1.2 : IDEM 1.1
        x_1 = math.floor(np.random.uniform(-bound, bound))
        x_2 = math.floor(np.random.uniform(-bound, bound))
        while(x_1 == x_2):
            x_1 = math.floor(np.random.uniform(-bound, bound))
            x_2 = math.floor(np.random.uniform(-bound, bound))

        y_1 = y_2 = math.floor(np.random.uniform(-bound, bound))
```

```
75          if ((x_1, y_1, x_2, y_2) not in segments):
76              segments.add((x_1, y_1, x_2, y_2))
77              hs_ctr += 1
78          if (hs_ctr == hs_qty):
79              break
80
81      segments = list(segments)
82      ## If we don't shuffle, then the segments list would end up with all the
83      ##        vertical segments first and then all the horizontal ones.
84      np.random.shuffle(segments)
85      return segments
86
87  generate_files()
```

7.1.2. Ordenamiento de Datos

```java
/*
* For this script to run correctly one must have a folder in the same directory filled with the input files.
* The name of this folder is passed in as the 3 argument.
* Usage: Main B M InputsFolder CoordinateToSortBy
* Last test : 21/05/2017
* Arguments : 40000 400000 Inputs x
* This means, B = 4000 bytes | M = 40000 | inputs folder = Input
*
* */

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.*;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import static java.nio.file.StandardOpenOption.APPEND;
import static java.nio.file.StandardOpenOption.CREATE;

public class Main {


    static int B;
    static int M;
    static int m;
    static Random nameGenerator = new Random();
    static int numberOfDiskIO = 0;

    public static void main(String[] args) throws IOException{

        // We receive as an argument the directory where we previously stored all the generated inputs.
        Path inputsFolderPath = Paths.get(args[2]);
        // Path to output file where we will write the following:
        // Input-Name.csv , Number of disk IO's.
        Path resultPath = Paths.get("resultsy.csv");

        B = Integer.parseInt(args[0]);
        M = Integer.parseInt(args[1]);
        m = (int)Math.ceil(M/B);
        char coordToSort = 'y';


        // Here we store the paths to all input files.
        List<Path> inputPaths;
        try (Stream<Path> paths = Files.walk(inputsFolderPath)) {
            inputPaths = paths.filter(Files::isRegularFile).collect(Collectors.toList());
        }

        // We iterate through the input files, calling mergeSort on each one of them and counting
        // the number of disk IO's.
        for (Path aPath: inputPaths) {
            numberOfDiskIO = 0;
            List<Path> sortedFilePath = externalMergeSort(aPath, coordToSort);
            // If after de MergeSort procedure, specifically after the merge part we have more than one
            // sorted run, it means something went wrong with the merge.
            if(sortedFilePath.size() != 1){
                System.out.println("ERROR: Merge ended with more than one resulting sorted run.");
                break;
            }
            // Construct the line to be written to the file.
            String line = String.format(aPath.getFileName().toString()+","+"%d"+System.lineSeparator(), ↩
                numberOfDiskIO);
            // Write to it.
            Files.write(resultPath, line.getBytes(), CREATE, APPEND);
        }


    }
    /*
    *  External MergeSort procedure. Given a path to the input file to be sorted and the coordinate to it by↩
       it
    *  proceeds to run the External MergeSort procedure. This complies of the following:
    *  1. Phase one: Generate n/m sorted runs and write them to secondary memory.
    *  2. Phase two: At each step merge m sorted runs into one.
    *     Repeat this last step until there is only one sorted run.
    *
    *  Path inputPath : Path to the input file to be sorted.
    *  char coordToSort : coordinate to sort the file by. (can be 'x' or 'y')
    * */
```

```
82      public static List<Path> externalMergeSort(Path inputPath, char coordToSort) throws IOException{
83          FileChannel anInput = FileChannel.open(inputPath);
84          List<Path> phaseOnePaths = phaseOne(anInput, coordToSort);
85          phaseOnePaths = phaseTwo(phaseOnePaths, coordToSort);
86          return phaseOnePaths;
87      }
88
89      /*
90      * Phase one of the external mergeSort algorithm. It takes an input file and produces n/m sorted runs which are
91      * stored in secondary memory for the second phase of the algorithm (Merge). These runs are sorted by the coordinate
92      * indicated by the argument coordToSort.
93      *
94      * FileChannel input : input file.
95      * char coordToSort : coordinate to sort the file by. (can be 'x' or 'y')
96      *
97      * Returns : List of paths to the files where each sorted run was written to.
98      * */
99      public static List<Path> phaseOne(FileChannel input, char coordToSort) throws IOException{
100         List<Path> sortedRuns = new ArrayList<>();
101         List<String> linesToWrite;
102         List<int[]> segmentsRun;
103
104         while(Math.abs(input.position() - input.size()) > 1){
105
106             segmentsRun = getSegmentsRun(input);
107             sortSegmentsRun(segmentsRun, coordToSort);
108
109             linesToWrite = segmentsRunToString(segmentsRun);
110
111             Path path = generateSegmentsRunName();
112             sortedRuns.add(path);
113
114             writeSortedRunToSM(linesToWrite, path);
115         }
116
117         return sortedRuns;
118     }
119
120     /*
121     * Phase two of the external mergeSort algorithm. It takes a list of paths to sorted runs and performs phase two steps
122     * until there is only one path in the lists of paths. The amount of paths decreases because at each step of the phase two
123     * we merge O(m) sorted runs into one large sorted run.
124     *
125     * List<Path> phaseOneOutput : Phase one output, ie, a list of paths to the sorted run generated by phaseOne method.
126     * char coordToSort : coordinate to sort the file by. (can be 'x' or 'y')
127     * */
128     public static List<Path> phaseTwo(List<Path> phaseOneOutput, char coordToSort) throws IOException{
129         List<Path> output = phaseOneOutput;
130         while(output.size() != 1){
131             output = phaseTwoStep(output, coordToSort);
132         }
133
134         return output;
135
136     }
137
138     /*
139     * A phase two step consists basically in merging O(m) sorted runs into a single one in secondary memory.
140     * This is done by taking the first m sorted runs in the sortedRuns list and continuously selecting the minimum
141     * between each one of them and storing it in an output buffer. Once this buffer reaches a size of B we flush it
142     * to secondary memory.
143     *
144     * List<Path> sortedRuns : List containing paths to each sorted run generated previously by another phase two step
145     * or the phase one. The idea here is to take m of these and after the merge is completed we delete them.
146     * char coordToSort : coordinate to sort the file by. (can be 'x' or 'y')
147     * */
148     public static List<Path> phaseTwoStep(List<Path> sortedRuns, char coordToSort) throws IOException{
149         List<ArrayDeque> inputBuffers = new ArrayList<>();
150         List<FileChannel> inputFiles = new ArrayList<>();
151         /* Path to file were we will store our generated merged run */
152         Path outputFilePath = generateSegmentsRunName();
153
154         /* This for loop basically initializes the input buffers and files. */
155         for (Path pathToInput: sortedRuns) {
156             /* We need to break from the loop when we have m - 1 input buffers (O(m)) */
157             if(inputBuffers.size() == m - 1){
158                 break;
159             }
160             /* FileChannel input to allow us to read from the file from disk whenever its corresponding inputbuffer is empty. */
```

```
161             FileChannel input = FileChannel.open(pathToInput);
162             /* The next two lines are the actual stuffing of the input buffer. */
163             List<int[]> segments = segmentsRunToInt(readBlockFromFile(input,B));
164             ArrayDeque segmentsQueue = new ArrayDeque(segments);
165             /* Now we add the input buffer to the input buffers list and the Filechannel to the files list. */
166             inputBuffers.add(segmentsQueue);
167             inputFiles.add(input);
168         }
169
170         /* We need to keep track of the size of the output buffer so we know when to perform a write to disk. */
171         int outputBufferSize = 0;
172         List<String> outputBuffer = new ArrayList<>();
173
174         /* Iterate until we have consumed all the input buffers. */
175         while(inputBuffers.size() > 0){
176
177             /* Before anything, we need to check if the output buffer is full (= size ~ B).
178             *  If it is we need to perform a write to disk operation.
179             *  */
180
181             if(outputBufferSize < B){
182                 /* We get the index associated with the input buffer which contains the minimum segment. */
183                 int minLocation = getInputBuffersMinLocation(inputBuffers, new SegmentComparator(coordToSort));
184
185                 /* Typical parsing. */
186                 int [] segment = (int[])inputBuffers.get(minLocation).removeFirst();
187                 String minSegment = String.format("%d,%d,%d,%d", segment[0], segment[1], segment[2], segment[3]);
188
189                 outputBuffer.add(minSegment);
190                 outputBufferSize += minSegment.getBytes().length;
191
192
193                 /* This if checks whether the input buffer from which we took the minimum is now empty or not. */
194                 if(inputBuffers.get(minLocation).size() == 0){
195                     /* If there is anything left in the corresponding input file then we load a chunk of data to the input buffer. */
196                     if(Math.abs(inputFiles.get(minLocation).position() - inputFiles.get(minLocation).size()) > 1){
197                         inputBuffers.get(minLocation).addAll(segmentsRunToInt(readBlockFromFile(inputFiles.get(minLocation), B)));
198                     }
199                     /* Else, it means that the input buffer is empty and the corresponding file has nothing new to read from. Thus
200                     *  we have reached the end of that sorted run and we can delete it. */
201                     else{
202                       inputBuffers.remove(minLocation);
203                       Files.delete(sortedRuns.get(minLocation));
204                       inputFiles.remove(minLocation);
205                       sortedRuns.remove(minLocation);
206                     }
207                 }
208
209             } else{
210                 writeSortedRunToSM(outputBuffer,outputFilePath);
211                 outputBuffer.clear();
212                 outputBufferSize = 0;
213             }
214
215         }
216
217         /* Before returning we write the remaining segments in the output buffer to the output file. */
218         writeSortedRunToSM(outputBuffer,outputFilePath);
219
220
221         /* We add the newly created sorted run -its path- (product of merging O(m) sorted runs) to the lists of sorted runs. */
222         sortedRuns.add(outputFilePath);
223
224         return sortedRuns;
225     }
226
227     /*
228     * Takes a list of queues where each one of them represents an input buffer of a sorted run. This method
229     * peeks the first elements of each buffer and returns the index of the list the contains the minimum.
230     *
231     * List<ArrayDeque> inputBuffers : List of queues representing input buffers.
232     * Comparator segmentComparator : segmentComparator class instance used to compare each segment and decide the minimum.
233     * */
234     public static int getInputBuffersMinLocation(List<ArrayDeque> inputBuffers, Comparator segmentComparator){
235         int [] min = {Integer.MAX_VALUE, Integer.MAX_VALUE, Integer.MAX_VALUE, Integer.MAX_VALUE};
236         int index = 0;
```

```
237         for (Deque segmentsDeque: inputBuffers) {
238             int [] segment = (int[]) segmentsDeque.getFirst();
239             if(segmentComparator.compare(min, segment) == 1){
240                 min = segment;
241                 index = inputBuffers.indexOf(segmentsDeque);
242             }
243         }
244 
245         return index;
246     }
247 
248     /*
249     * Takes a List of segments, where each segment is an array of integers of the form [x_1, y_1, x_2, y_2]
250     * and returns a List of Strings where each component is a string of the form "x_1,y_1,x_2,y_2".
251     *
252     * List<int[]> segmentsRun : List of segments to "convert".
253     * Returns : List of segments in String representation explained above.
254     * */
255     public static List<String> segmentsRunToString(List<int[]> segmentsRun){
256         List <String> linesToWrite = new ArrayList<>();
257         for (int[] segment: segmentsRun) {
258             String line = String.format("%d,%d,%d,%d", segment[0], segment[1], segment[2], segment[3]);
259             linesToWrite.add(line);
260         }
261         return linesToWrite;
262     }
263 
264     /*
265     * Takes a List of segments, where each segment is a string of the form "x_1,y_1,x_2,y_2"
266     * and returns a List of int[] where each component of the list is an array of ints of the form [x_1, y_1↩
         , x_2, y_2].
267     *
268     * List<String> segmentsRun : List of segments to "convert".
269     * Returns : List of segments in int[] representation explained above.
270     * */
271     public static List<int[]> segmentsRunToInt(List<String> segmentsRun){
272         List <int[]> segments = new ArrayList<>();
273         for (String line: segmentsRun) {
274             int [] segment = Stream.of(line.split(",")).mapToInt(Integer::parseInt).toArray();
275             segments.add(segment);
276         }
277         return segments;
278     }
279 
280     /*
281     * Generates a name for storing a sorted run in secondary memory.
282     * To generate random names it used nameGenerator, which is an instance of Random.
283     * Returns : Path containing name where a sorted run will be stored in secondary memory.
284     * */
285     public static Path generateSegmentsRunName(){
286         return Paths.get(String.format("%d.csv", nameGenerator.nextInt()));
287     }
288 
289     /*
290     * Given an input File of size N, we wish to take chunks of size B from it and start the sorting process.
291     * This method returns a buffer filled with the segments inside the chunk of size B of the input file.
292     * FileChannel input : Input file. Using a FileChannel allows us read B bytes from the file.
293     * int B : Size of chunks we read from the file (bytes).
294     *
295     * Returns : List<String> where each component is a String representing a segment. Each String looks like↩
          "x_1,y_1,x_2,y_2"
296     * */
297     public static List<String> readBlockFromFile(FileChannel input, int B) throws IOException {
298 
299         /*
300         * A big problem in reading text files by a certain amount of bytes is dealing with line cuts.
301         * It could happen that we wish to read X bytes from the file and that means reading a non integer
302         * amount of lines. In this case we need to rewind the file pointer to the last \n read. So we can ↩
              start
303         * from there the next time.
304         * */
305 
306         numberOfDiskIO++;
307 
308         boolean EOBuffer = false;
309         // Here we create our byte buffer with a fixed size B.
310         ByteBuffer byteBuffer = ByteBuffer.allocate(B);
311         // We read from the file. Starting from the last position we were in.
312         input.read(byteBuffer);
313         // To read from the buffer we need to move the pointer position to the beggining.
314         byteBuffer.flip();
315 
316         // Lines is the list in which we will store the segments read from the chunk of size B.
317         List<String> lines = new ArrayList<>();
318         String lineString = "";
319         char c;
320 
321         // We read from the buffer while it still has bytes to read.
```

```
322         while(byteBuffer.hasRemaining()){
323 
324             // We get the first char from the buffer and check if it is a end of line.
325             c = (char)byteBuffer.get();
326             if(c == System.lineSeparator().charAt(0) && !byteBuffer.hasRemaining()){
327                 EOBuffer = true;
328             }
329             // This case happens when the last chunk read left us at the end of a line.
330             if(c == System.lineSeparator().charAt(0) && byteBuffer.hasRemaining()){
331                 c = (char)byteBuffer.get();
332             }
333             while(c != System.lineSeparator().charAt(0)){
334                 lineString += c;
335                 /*
336                 * While we don't see an end of line we keep on reading the line. Mind the fact that if we ↩
                        are in a line
337                 * cutted by the read method we would never see an end of line.
338                 * For this, if the bytebuffer does not have any more bytes to get and we haven't reached and↩
                        end of line
339                 * it means we need to stop reading.
340                 * */
341                 if(!byteBuffer.hasRemaining()){
342                     EOBuffer = true;
343                     break;
344                 }
345                 c = (char)byteBuffer.get();
346             }
347             if(EOBuffer){
348                 break;
349             }
350             /*
351              * This one-liner simply gets the lineString of the form "x1,y1,x2,y2", strips it by ',' and ↩
                    converts each
352              * component to an integer, then an array of integers.
353             */
354             //System.out.println(lineString);
355             //segments = Stream.of(lineString.split(",")).mapToInt(Integer::parseInt).toArray();
356             lines.add(lineString);
357             lineString = "";
358         }
359 
360 
361         /*
362         * Now if we ended up in the middle of a line we need to calculate how much we need to rewing the ↩
                input file
363         * position, so we can read the whole line the next time.
364         * */
365         int fileRewind = 0;
366         int bufferPosition = byteBuffer.position()-1;
367         char lastCharRead = (char)byteBuffer.get(bufferPosition);
368         if(lastCharRead != System.lineSeparator().charAt(0)){
369             bufferPosition--;
370             lastCharRead = (char)byteBuffer.get(bufferPosition);
371             while(lastCharRead != System.lineSeparator().charAt(0)){
372                 fileRewind++;
373                 bufferPosition--;
374                 lastCharRead = (char)byteBuffer.get(bufferPosition);
375             }
376         }
377         // Finally, we update de input file's position to the end of the last line we were able to read ↩
                completely.
378         input.position(input.position() - fileRewind - 1);
379 
380 
381         return lines;
382     }
383 
384     /*
385     *  Given segmentsRun with each component inside of it of the form [x_1, y_1, x_2, y_2]
386     *  it uses the SegmentComparator to sort the list given a coordinate (coordToSort)
387     *
388     *  int [] segmentsRun : a run of unordered segments.
389     *  char coordtoSort : coordinate to sort by segments. (Values can be 'x' or 'y')
390     *
391     *  Its void since the sort is done in-place.
392     * */
393     public static void sortSegmentsRun(List<int[]> segmentsRun, char coordToSort){
394         Collections.sort(segmentsRun, new SegmentComparator(coordToSort));
395     }
396 
397     /*
398     * Reads m times blocks of size B from the input. Each block is a set of lines in the input.
399     * It puts them all together in the segmentRun List.
400     *
401     * FileChannel input: Input file.
402     *
403     * Returns: List<int[]> of segments. Each component of the list a segment represented in an array of ints
404     * of the form [x_1, y_1, x_2, y_2]
```

```
405     * */
406     public static List<int[]> getSegmentsRun(FileChannel input) throws  IOException{
407         List<String> blockReadFromFile;
408         List<int[]> segmentsRun = new ArrayList<>();
409 
410         for (int i = 0; i < m; i++) {
411             blockReadFromFile = readBlockFromFile(input, B);
412             for (String line: blockReadFromFile) {
413                 int [] segment = Stream.of(line.split(",")).mapToInt(Integer::parseInt).toArray();
414                 segmentsRun.add(segment);
415             }
416         }
417 
418         return segmentsRun;
419     }
420 
421     /*
422     * Gets a sorted run of segments and writes it to secondary memory in the specified path.
423     * List<String> sortedRun : List of sorted segments (single run).
424     * Path path : Path to file in which the sorted run is going to be written.
425     * */
426     public static void writeSortedRunToSM(List<String> sortedRun, Path path) throws IOException{
427         numberOfDiskIO++;
428         Files.write(path, sortedRun,CREATE,APPEND);
429     }
430 
431     /*
432     * Comparation function for sorting the random sample in getPivots.
433     * Constructor: SegmentComparator
434     * */
435     static private class SegmentComparator implements Comparator<int[]> {
436 
437         char cts;
438         /*
439         * Constructor for the class Comparator.
440         * coordToSort : coordinate by which we are sorting the file (x or y).
441         * */
442         private SegmentComparator(char coordToSort){
443             cts = coordToSort;
444         }
445 
446         /*
447         * segmentOne : First segment involved in comparison.
448         * segmentTwo : Second segment involved in comparison.
449         * */
450         public int compare(int[] segmentOne, int[] segmentTwo){
451             // Notice that the homework pdf states that there a some difference in sorting by x or y.
452             if (cts == 'x'){
453                 if (Integer.compare(segmentOne[0], segmentTwo[0]) == 0){
454                     return Integer.compare(segmentOne[2], segmentTwo[2]);
455                 }else{
456                     return Integer.compare(segmentOne[0], segmentTwo[0]);
457                 }
458 
459             }
460 
461             // Here we need to check if a segment is vertical or horizontal to decide ties between segments.
462             if (cts == 'y'){
463 
464                 // Case 1: The two segments both have the same y coordinates, ie:
465                 //  S1_y1 = S2_y1 and S1_y2 = S2_y2
466                 if (Integer.compare(segmentOne[1], segmentTwo[1]) == 0 && Integer.compare(segmentOne[3], ↩
                        segmentTwo[3]) == 0){
467 
468                     // Note that a segment is vertical if both of its y coordinates are the same.
469                     // So if Segment one is vertical, we claim it larger by comparison.
470                     if(Integer.compare(segmentOne[0], segmentOne[2]) == 0){
471                         return 1;
472                     }else{
473                         return -1;
474                     }
475                     // Case 2: The two segments differ in at least one y coordinate.
476                 }else{
477                     if (Integer.compare(segmentOne[1], segmentTwo[1]) == 0){
478                         return Integer.compare(segmentOne[3], segmentTwo[3]);
479                     }else{
480                         return Integer.compare(segmentOne[1], segmentTwo[1]);
481                     }
482 
483                 }
484             }
485 
486             return 0;
487         }
488     }
489 
490 }
```

7.1.3. Graficar resultados

```python
import csv
import matplotlib.pyplot as plt


# Auxiliary function to get the second value of a tuple
def getKey(item):
    return item[1]


# Ploter for the results
def plot(coord):
    data = open('results'+coord+'.csv', 'rb')
    reader = csv.reader(data, delimiter=',')
    uniform25 = list()
    normal25 = list()
    uniform50 = list()
    normal50 = list()
    uniform75 = list()
    normal75 = list()

    # Separate the results in corresponding distribution and proportion
    for row in reader:
        file = row[0]
        value = row[1]
        att = file.split('-')
        if att[3] == "normal":
            if att[4] == "0.25.csv":
                normal25.append([int(value), int(att[2])])
            elif att[4] == "0.5.csv":
                normal50.append([int(value), int(att[2])])
            else:
                normal75.append([int(value), int(att[2])])
        else:
            if att[4] == "0.25.csv":
                uniform25.append([int(value), int(att[2])])
            elif att[4] == "0.5.csv":
                uniform50.append([int(value), int(att[2])])
            else:
                uniform75.append([int(value), int(att[2])])

    #Remove repeated values
    old_uniform25 = uniform25
    uniform25 = []
    old_uniform50 = uniform50
    uniform50 = []
    old_uniform75 = uniform75
    uniform75 = []

    old_normal25 = normal25
    normal25 = []
    old_normal50 = normal50
    normal50 = []
    old_normal75 = normal75
    normal75 = []

    for x in old_uniform25:
        if x not in uniform25:
            uniform25.append(x)

    for x in old_uniform50:
        if x not in uniform50:
            uniform50.append(x)

    for x in old_uniform75:
        if x not in uniform75:
            uniform75.append(x)

    for x in old_normal25:
        if x not in normal25:
            normal25.append(x)

    for x in old_normal50:
        if x not in normal50:
            normal50.append(x)

    for x in old_normal75:
        if x not in normal75:
            normal75.append(x)

    # Sort values and only save accesses
    uniform25 = [x[0] for x in sorted(uniform25, key=getKey)]
    uniform50 = [x[0] for x in sorted(uniform50, key=getKey)]
    uniform75 = [x[0] for x in sorted(uniform75, key=getKey)]
```

```
84      normal25 = [x[0] for x in sorted(normal25, key=getKey)]
85      normal50 = [x[0] for x in sorted(normal50, key=getKey)]
86      normal75 = [x[0] for x in sorted(normal75, key=getKey)]
87      numbers = range(9, 22)
88
89      # Plot comparing all the tests
90      plt.plot(numbers, normal25, 'b—', label='Normal 0.25')
91      plt.plot(numbers, normal50, 'bs', label='Normal 0.50')
92      plt.plot(numbers, normal75, 'b^', label='Normal 0.75')
93      plt.plot(numbers, uniform25, 'g—', label='Uniforme 0.25')
94      plt.plot(numbers, uniform50, 'gs', label='Uniforme 0.50')
95      plt.plot(numbers, uniform25, 'g^', label='Uniforme 0.75')
96      plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=2, mode="expand", borderaxespad=0.)
97      plt.xlabel('Orden de potencia de 2 de segmentos')
98      plt.ylabel('Numero de acceso a discos')
99      plt.savefig('all'+coord+'.png', bbox_inches='tight')
100     plt.clf()
101
102     # Plot comparing test with alpha = 0.25
103     plt.plot(numbers, normal25, 'b—', label='Normal 0.25')
104     plt.plot(numbers, uniform25, 'g—', label='Uniforme 0.25')
105     plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=2, mode="expand", borderaxespad=0.)
106     plt.xlabel('Orden de potencia de 2 de segmentos')
107     plt.ylabel('Numero de acceso a discos')
108     plt.savefig('25'+coord+'.png')
109     plt.clf()
110
111     # Plot comparing test with alpha = 0.50
112     plt.plot(numbers, normal50, 'b—', label='Normal 0.50')
113     plt.plot(numbers, uniform50, 'g—', label='Uniforme 0.50')
114     plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=2, mode="expand", borderaxespad=0.)
115     plt.xlabel('Orden de potencia de 2 de segmentos')
116     plt.ylabel('Numero de acceso a discos')
117     plt.savefig('50'+coord+'.png')
118     plt.clf()
119
120     # Plot comparing test with alpha = 0.75
121     plt.plot(numbers, normal75, 'b—', label='Normal 0.75')
122     plt.plot(numbers, uniform75, 'g—', label='Uniforme 0.75')
123     plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=2, mode="expand", borderaxespad=0.)
124     plt.xlabel('Orden de potencia de 2 de segmentos')
125     plt.ylabel('Numero de acceso a discos')
126     plt.savefig('75'+coord+'.png')
127     plt.clf()
128
129 # Plot for x and y tests
130 plot('x')
131 plot('y')
```