Ingeniería en ciencias de la computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
CC4102-1 Diseño y Análisis de Algoritmos

TAREA DISEÑO Y ANÁLISIS DE ALGORITMOS

Tarea N°3

Integrantes: Diego Echeverría Pentzke
Agustin Muñoz Peralta
Profesor: Gonzalo Navarro
Auxiliares: Asunción Gómez
Diego Salas
Fecha de entrega: 7 de julio de 2023
Santiago, Chile

Índice de Contenidos

Lista de Figuras

Lista de Tablas

Lista de Códigos

1. Introducción

Para este informe se tiene la motivación de comparar la eficiencia en búsquedas al momento de implementar un filtro bloom sobre nombres en una base de datos. Además, se querrá ver como cambiar los parametros de este filtro afecta en su probabilidad de falsos positivos.

Para esto, se implementará un filtro Bloom sobre un archivo csv que contiene nombres comunes para luego buscar en este archivo según el filtro, comparando con una busqueda directa sin filtrar.

Para nuestra hipótesis inicial se cree que al implementar un filtro bloom, sin importar el caso, mejora el tiempo de busqueda sobre los datos. Esto pues, el filtro debiese prevenir correctamente los casos de busqueda innecesarios y solo ejecutar la busqueda cuando un elemento se encontrará de manera efectiva dentro de la base de datos. Cabe mencionar que esto **no** mejorará el tiempo de busqueda como tal, sino los tiempos de busqueda sobre un arreglo con nombres efectivos en la base de datos y strings que no pertenezcan.

Para poner a prueba la hipótesis, se realizarán múltiples tests con variables:

- N = $[2^{16}, 2^{17}, 2^{18}]$
- M = $[5 \cdot N, 10 \cdot N, 20 \cdot N]$ bits
- k = $\frac{M}{N} * 0{,}5, \frac{M}{N} * \ln(2), \frac{M}{N}$

En donde 'N' indica la cantidad de elementos a buscar, 'M' indica la cantidad de bits utilizados en el filtro Bloom y 'k' la cantidad de funciones *hash* a utilizar en el filtro.

Estos valores fueron elegidos luego de una investigación sobre el filtro Bloom y sus parametros. Para N, simplemente se eligen tamaños cercanos al total de elementos en la base de datos, al rededor de las 98000 entradas, luego se toma una variacion superior e inferior.
Para los valores de M, se considera suficiente el valor $M = 10 \cdot N$, pues, según Wikipedia, con a lo más 10 bits por elemento guardado se consigue una probabilidad de falsos positivos menor a un 1 %, valor que consideramos óptimo y aceptable para este experimento. Luego de establecida esta condición, variamos el valor de forma superior e inferior.
Análogo a M, para los valores de 'k' se considera en la misma fuente que un valor óptimo es $k = \frac{M}{N} \cdot \ln(2)$ pues minimiza el porcentaje de falsos positivos. Finalmente se toman valores menores y mayores variando su constante.

Luego, consideramos como valores optimos el par (M, k) tal que M y k son sus valores optimos.

Se espera también, dada la naturaleza del filtro Bloom, que ocupe un espacio O(M), pues cada elemento se guardará con M bits. Sobre su tiempo, creemos y nos apoyamos en la fuente ya mencionada para predecir que tendrá un tiempo O(k), pues el filtro depende directamente de la ejecución de nuestras 'k' funciones hash.

El tiempo de ejecución será medido a través de la librería *timeit* de Python. De esta forma, podremos saber en milisegundos el tiempo de cada algoritmo y se podrá comparar de forma empírica.

Con los resultados obtenidos se graficarán los tiempos, se analizarán y se discutirá sobre estos resultados para así llegar a una conclusión sobre nuestra hipótesis y el rendimiento de ambas estructuras en distintos escenarios.

2. Desarrollo

Para la ejecución de nuestro experimento, se considera que, para cada tamaño de búsqueda N, probar cada par posible (M, k) tal que podamos observar como cada combinación afecta a nuestra tasa de falsos positivos. Además, en cada una de estas búsquedas se medirá el tiempo que será compara con su contra parte de búsqueda sin filtro.

Además, por cada búsqueda utilizando el test, se buscará en el csv de forma independiente, teniendo la misma cantidad de resultados con y sin filtro de Bloom.

2.1. Implementación

2.1.1. Filtro de Bloom

Para la implementación de nuestro filtro de Bloom, se utiliza como base un código recuperado desde la pagina GeeksForGeeks en Python. Luego de hacer modificaciones para que calzara con nuestras necesidades, queda como sigue:

Código 1: Filtro de Bloom

```
class BloomFilter(object):
    def ___init___(self, m, primes, k):
        # Size of bit array to use
        self.size = m

        # number of hash functions to use
        self.hash_count = k

        # Bit array of given size
        self.bit_array = bitarray(self.size)

        # initialize all bits as 0
        self.bit_array.setall(0)

        #initialize prime numbers
        self.primes = primes
```

Aquí, se inicializa el filtro de Bloom, con nuestro valor de M bits, cuantas k funciones *hash* a utilizar y un arreglo de primos que será explicado mas adelante.

Código 2: Agregar al filtro de bloom

```
    def add(self, string):
        digests = []

        # Cada de estos es un hash distinto
        for i in range(self.hash_count):

            # primes[i] permite crear distintos hashes
            digest = self.hash(string, self.primes[i]) % self.size
            digests.append(digest)

```

```
11          # set the bit True in bit_array
12          self.bit_array[digest] = True
```

En este segundo bloque, la implementación de como añadir un string a nuestro filtro es a través de un ciclo for. Con esto se toma un string, se le aplican las k funciones *hash* y cada una de ellas devuelve un bit el cual debe ser activado, esto para cada función, por lo que cada string terminará marcando k bits.

Este funcionamiento es análogo para la búsqueda, se calcula cada *hash* del string y luego se revisan los bits correspondientes.

Finalmente, se implementa nuestra función *hash* escogida. En particular se decidió por utilizar un polynomial Rolling Hash para strings, recuperado del sitio GeeksForGeeks. Esto pues es una función *hash* de amplio uso a través de la computación (Diseño y Análisis de Algoritmos Apuntes, p. 122, Gonzalo Navarro) y fácil de variar al utilizar distintos números primos, es por esto que nuestro filtro de Bloom recibe un arreglo de k primos, siendo cada uno de estos utilizado para una nueva funcion de *hash*.

Código 3: Funciones de hashing

```
1   def hash(self, string, p):
2       # total
3       sum = 0
4       # valor de m
5       m = 10**9 + 7
6       # potencia de p, p^i
7       p_pow = 1
8       # Para cada caracter, calculamos el aporte al hash
9       for i in range(len(string)):
10          sum = (sum + (1 + ord(string[i]) - ord('a')) * p_pow) % m
11          p_pow = (p_pow * p) % m
12
13      return sum
```

Con esto, al variar el numero primo p se pueden conseguir las k funciones *hash* distintas. Cabe mencionar que el valor m no es nuestra cantidad de bits, sino un entero utilizado comúnmente para esta función *hash*.

Una vez implementado el filtro de Bloom, pasamos a la implementación del experimento.

2.1.2. Experimento

La parte importante de esta implementación, recae en nuestra funcion *bloomTest(size, m, k)* que recibe tres parametros. Estos corresponden al largo de busqueda, la cantidad de bits por elementos y un factor en la función $\frac{M}{N} \cdot factor$. Por lo que, con un ciclo for y una matriz de pares, se ejecuta la función *bloomTest(size, m, k)* para cada par, para cada tamaño N. Esta función se explicara por partes como sigue.

Código 4: Inicialización del filtro

```
1 def bloomTest(size, _m, _k):
2   N = 1 << size # Busquedas
3
```

```
babyNames = csv.reader(open('Popular-Baby-Names-Final.csv', "r", encoding='utf8'), delimiter=","
    )

# Resta nombre de columna
lenBabyNames = sum(1 for row in babyNames) - 1

# Calculo bits y cantidad de funciones
m = _m * lenBabyNames # Bits
k = math.ceil(_k)

# Genero k primos, uno por cada hash
primeFloor = 29791 # Numero muy utilizado en progcomp para este hash
primeBound = 30030 # Con el maximo k no pasa de este numero
primesList = pr.between(primeFloor, primeBound)
primesList = primesList[0:k]

# Inicializar bloom filter
bloom = bf(m, primesList, k)
```

En la primera sección, se calculan e inicializan las variables necesarias para el filtro, notemos la generación de primos utilizando la librería primePy de Python, esta permite la fácil generación de k números primos. Los valores *primeFloor* y *primeBound* son debidos a la naturaleza de el *hash* utilizado, en donde el primero es un valor comúnmente utilizado en el ámbito de programación competitiva y el segundo simplemente es un valor que no será alcanzado al utilizar la mayor combinación posible de (M, k), teniendo así una forma de generar los k primos para cualquier par.

Finalmente, se inicializa nuestro filtro Bloom con nuestros $m = M$ bits, una lista de k primos y k la cantidad de funciones.

Código 5: Creación de arreglo de busqueda

```
search = [] # arreglo de busqueda
alpha = 0.2 # razon entre nombres
moviesTotal = math.ceil(alpha * N) # cantidad de nombres de pelicula
namesTotal = N - moviesTotal # Cantidad de nombres
filmCount = 0 # Contador de peliculas
namesCount = 0 # Contador de nombres

# Mientras no llegue a mi total
while filmCount < moviesTotal:
  filmNames = csv.reader(open('Film-Names.csv', "r", encoding='utf8'), delimiter=",")
  for filmName in filmNames:
    # Evita posibles errores de NA
    if type(filmName[0]) != str:
      pass
    # Si llené lo suficiente, termino
    if filmCount == moviesTotal:
      break
    search.append(filmName[0])
    filmCount += 1

# Mientras no tenga suficientes, sigo llenando
while namesCount < namesTotal:
```

```
  babyNames = csv.reader(open('Popular-Baby-Names-Final.csv', "r", encoding='utf8'), delimiter=",
   ")
  for babyName in babyNames:
    # Evita errores de tipo NA
    if babyName[0] == 'Name' or type(babyName[0]) != str:
      pass
    if namesCount == namesTotal:
      break
    search.append(babyName[0])
    namesCount += 1
# Shuffle
random.shuffle(search)
```

En esta segunda sección, se inicializa el arreglo de búsqueda para nuestro experimento. Para esto, se decide utilizar un factor *alpha* que indica el porcentaje de nombres de película que tendrá el arreglo, esto para poder simular búsquedas infructuosas y así medir los falsos positivos. Se decide arbitrariamente que un 20 % del arreglo serán búsquedas de este estilo, para visualizar correctamente la diferencia entre el filtro Bloom y una búsqueda directa cuando se intenta buscar elementos que no existen en la base de datos.

Luego, se calcula la cantidad de elementos a ingresar, y con un ciclo for se pobla el arreglo de búsqueda según las cantidades calculadas. Finalmente, con la librería Random de Python, se genera una permutación aleatoria del arreglo, si bien esto no debiese afectar los tiempos de búsqueda se considera una buena medida para simular búsquedas en un ambiente real.

Código 6: Busqueda con Bloom

```
# * Busco en el bloom
negative = 0 # Cantidad de negativos del Bloom
falsePositive = 0  # Cantidad de positivos que en verdad no están

start = timer() # Inicio timer
for i in range(len(search)):
  name = search[i] # nombre a buscar
  inBloom = bloom.check(name) # Reviso si está en el filtro

  # Si está en filtro, busco
  if (inBloom):
    inNames = searchCsv(name) # bool

    # Si no encontre, entonces era una pelicula
    if not inNames:
      falsePositive += 1
  # Si no esta en filtro, cuento y sigo
  else:
    negative += 1

end = timer() # termino el timer
print('search done at: ', 1000 * (end - start), ' miliseconds')

# Calculamos la tasa de falsos positivos
fp = falsePositive / (falsePositive + negative)
print('La tasa de fp con m: ', m, ' y k: ', k, ' es: ', fp, '\n')
```

Notemos que, para calcular nuestra tasa de falsos positivos, debemos llevar un contador para la cantidad de negativos que indica el filtro Bloom, es decir, elementos que no se encuentran en la base de datos, y la cantidad de falsos positivos, es decir, la cantidad de elementos que dicen estar en la base de datos cuando en verdad no se encuentran. Luego, recorriendo el arreglo de búsqueda se revisa si según el filtro el elemento se encuentra o no. Si se encuentra el elemento en el filtro, se busca con la función *searchCsv(name)* que devuelve un valor booleano; si es *False* significa que se buscó un elemento que no estaba en la base de datos y se contabiliza como un falso positivo. Si no se encuentra el elemento en el filtro, se contabiliza un negativo.

Finalmente, se calcula el falso positivo como una razón tal que $FP = \frac{FalsePositives}{FalsePositives+TrueNegatives}$ y se imprime en pantalla.

Código 7: Búsqueda secuencial

```
start = timer()
for i in range(len(search)):
  searchCsv(search[i])
end = timer()

def searchCsv(name):
  df = csv.reader(open('Popular-Baby-Names-Final.csv', "r", encoding='utf8'), delimiter=",")
  for row in df:
    if row[0] == name:
      return True
```

La última sección de la implementación, es la búsqueda simple secuencial en el archivo csv con la función *searchCsv(name)*. Para cada elemento *name* en el arreglo de búsqueda, se recorre todo el csv para decidir si se encuentra o no. Notemos que al abrir el archivo por cada elemento se utiliza tiempo innecesario que es lo que se quiere evitar al utilizar el filtro de Bloom.

Así, se tiene una implementación del filtro Bloom y un experimento que permite comparar su uso en búsquedas con una búsqueda directa secuencial, además de poder calcular como varia su porcentaje de falsos positivos según la cantidad de bits y funciones *hash* a utilizar.

3. Resultados

La experimentación contempló los pares tal que:

M/k	0.5	ln(2)	1
5	5, 3	5, 4	5, 5
10	10, 5	10, 7	10, 10
20	20, 10	20, 14	20, 20

Cada uno de estos 9 pares fue ejecutado 3 veces, uno por cada posible tamaño de N.
Los resultados se tabulan como sigue.

3.1. Tiempos

Tomando el promedio de los tiempos con un M particular, se consigue que:

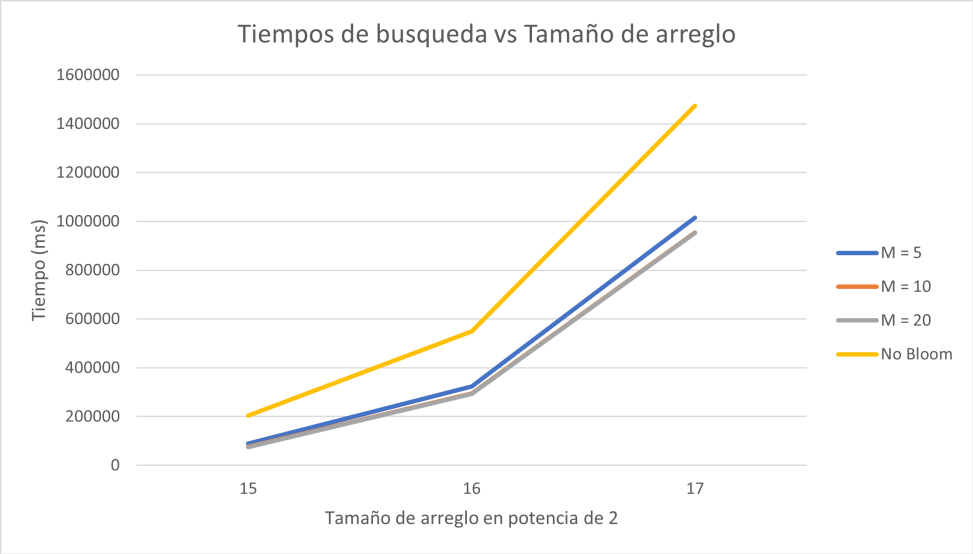

Figura 1: Tiempos promedio conseguidos según M

Esto es, para cada M, se promedian los valores obtenidos con los 3 valores de k posible, esto para cada valor de N.

Asimismo para un K particular:

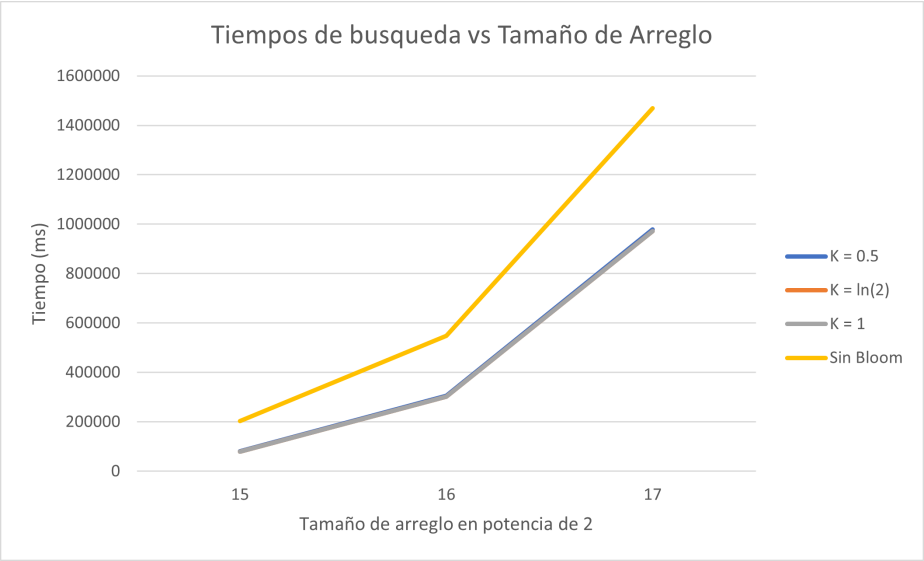

Figura 2: Tiempos promedio conseguidos según K

Luego, para cada par (M, k) en particular se consiguen las siguientes 9 comparaciones de tiempo:

3.1.1. M = 5

- $K = 0{,}5$

Figura 3: Comparativa par (5, 3)

- $K = \ln(2)$

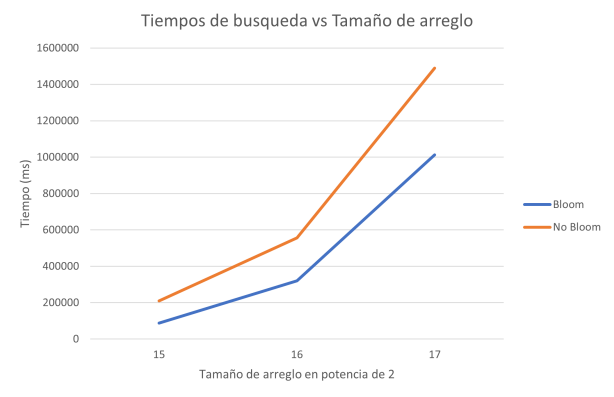

Figura 4: Comparativa par (5, 4)

- $K = 1$

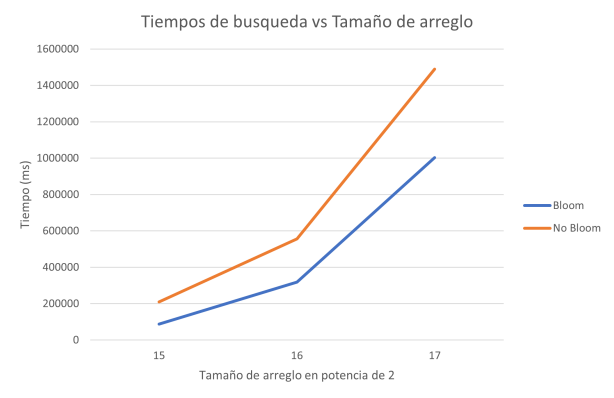

Figura 5: Comparativa par (5, 5)

3.1.2. M = 10

- $K = 0{,}5$

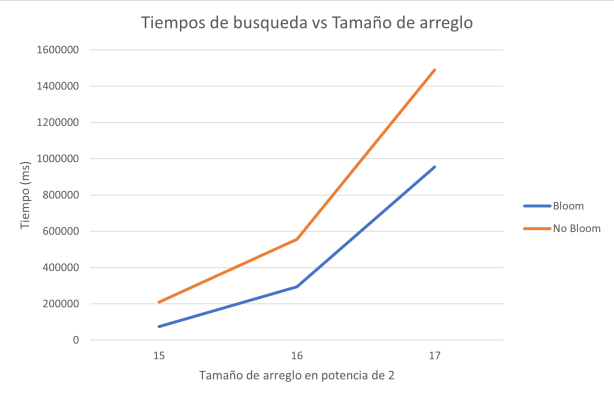

Figura 6: Comparativa par (10, 5)

- $K = \ln(2)$

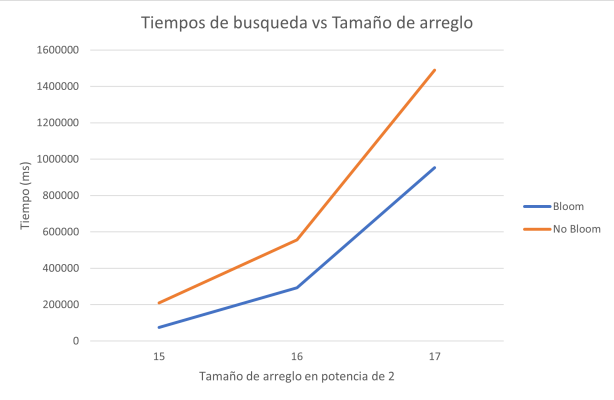

Figura 7: Comparativa par (10, 7)

- $K = 1$

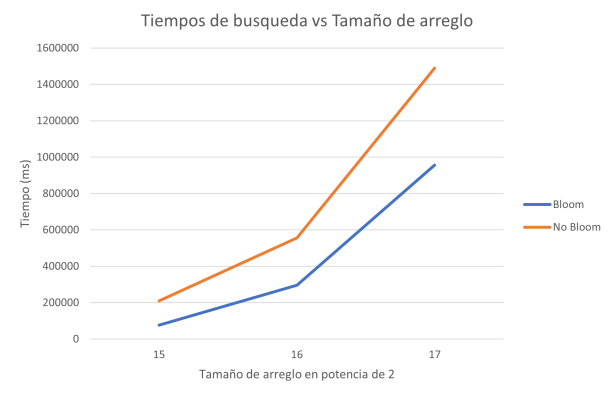

Figura 8: Comparativa par (10, 10)

3.1.3. M = 20

- $K = 0{,}5$

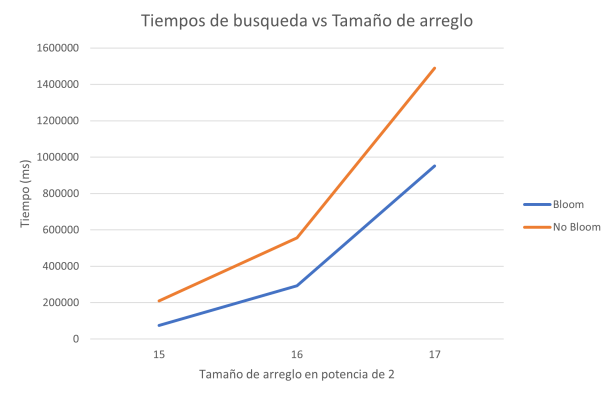

Figura 9: Comparativa par (20, 10)

- $K = \ln(2)$

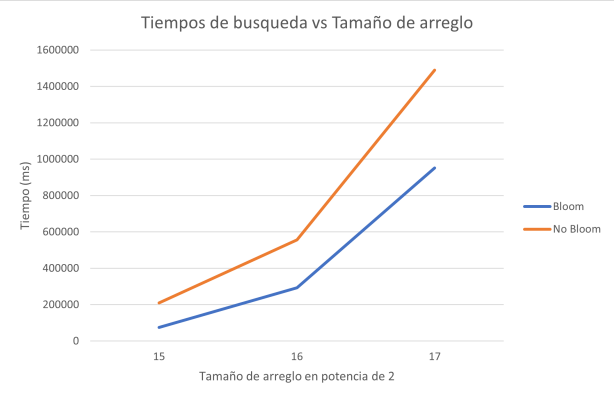

Figura 10: Comparativa par (20, 14)

- $K = 1$

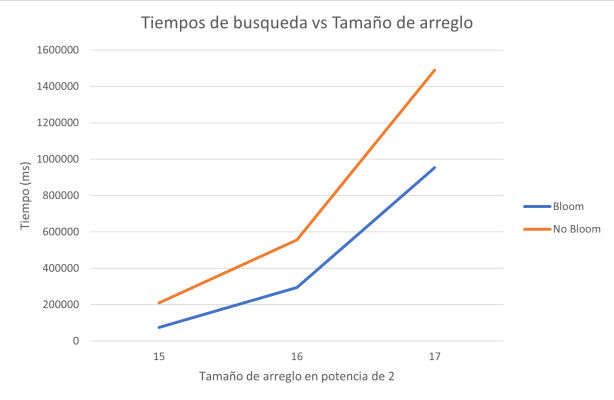

Figura 11: Comparativa par (20, 20)

3.2. Falsos Positivos

En esta sección, los resultados tabulados de cada par en relación a su porcentaje de falsos positivos es como sigue:

Tabla 1: Porcentajes de falsos positivos según pares (M, k)

M/k	0.5	ln(2)	1
5	0.0994	0.0939	0.1034
10	0.0085	0.0060	0.0096
20	0	0	0

4. Conclusión

Por los resultados es directo ver que, para toda configuración (M, k) utilizada, el filtro de Bloom reduce efectivamente el tiempo de búsqueda en una base de datos. Basta con ver los gráficos presentados en la sección anterior para notar la diferencia de magnitud entre el tiempo de búsqueda sin un filtro de Bloom y cualquier iteración que utilice este filtro.

También, es posible ver que no hay una diferencia sustancial entre los tiempos de búsqueda al variar los valores de K dentro de un mismo M.

Si se puede notar una diferencia entre los porcentaje de falsos positivos entre pares, en particular, se aprecia que un cambio en k no es tan crítico como un cambio en la cantidad de bits M, pues al ver la tabla anterior, notamos que los valores no varían en gran medida dentro de una misma fila, sin embargo al ver las columnas se ve una gran variación, llegando incluso a un valor cercano a 0.

Creemos que en pares con $M = 20$, el valor de porcentaje de falsos positivos es 0 debido a una excesiva cantidad de bits. Esto no es necesariamente bueno, puesto que con una gran cantidad de elementos en el filtro se malgastaría espacio en bits que tienen baja probabilidad de ser activados o buscados.

Tomando los valores de $M = 20$ como valores no representativos, debido a su ocupo innecesario de memoria, se puede ver como, para cualquier M, se consiguen los mejores resultados con un valor de $k = \frac{M}{N} \cdot \ln(2)$.

A partir de este análisis se puede concluir que, nuestra hipótesis en relación a los tiempos es correcta, pues en cualquier caso se obtuvo que el filtro Bloom mejora el tiempo de búsqueda en relación a búsquedas sin filtro.

Además, los resultados sobre como afecta k a los tiempos de búsqueda no es conclusivo, pues no se puede apreciar una variación sustancial que apoye un orden de O(k) en ejecución con filtro de Bloom.

Se tiene también que, con ciertas consideraciones de memoria, el óptimo valor de k es $k = \frac{M}{N} \cdot \ln(2)$. Esto es acorde con nuestra teoría según los valores teóricos encontrados en fuentes.

Sobre la memoria, es directo que, si cada elemento utiliza una cantidad M de bits, un filtro Bloom utiliza espacio de orden O(M) con elementos constantes.