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

Tarea 3

Algoritmos probabilísticos

Integrantes: Martín Paredes
Gonzalo Pérez
Carlos Stekel
Profesor: Gonzalo Navarro
Auxiliar: Asunción Gómez
Diego Salas
Ayudantes: Valeria Burgos
Nicolás Canales
Cristian Carrión
Gabriel Iturra
Yuval Linker
Camilo Rosas

Fecha de entrega: 7 de julio de 2023
Santiago de Chile

Índice de Contenidos

Índice de Figuras

Índice de Tablas

1. Introducción

En el siguiente informe se busca implementar un filtro de Bloom, el cual corresponde a un algoritmo probabilístico que sirve para buscar elementos en una base de datos, evitando recorrerla completa; con el fin de comparar los tiempos obtenidos en función de diversos parámetros. Este algoritmo probabilístico es *one-side error*, en que si el algoritmo dice que si el elemento buscado no está en la base, efectivamente no está, mientras que si dice que está podría equivocarse.

La hipótesis es que el costo en memoria al agregar el filtro de Bloom se espera que aumente en $O(m)$, en donde m corresponde al tamaño de la tabla de hash usada por dicho algoritmo, en que su valor óptimo teórico corresponde a:

$$m = -\frac{n \ln \epsilon}{(\ln 2)^2}$$

En donde n corresponde al tamaño de la base de datos y ϵ corresponde a la probabilidad de falsos positivos en el filtro. Dicha relación se obtiene como se detallará en el desarrollo. Nótese que considerando las constantes, principalmente se tendrá que dicho costo es $O(m) = O(n)$ pero es destacable que a medida que ϵ disminuye, el valor de m aumenta considerablemente. Asimismo la metodología a usar para obtener funciones de hash considerará valores de a y b repetidos k veces, con k el número de funciones de hash que usará cada filtro, en que su óptimo teórico es:

$$k = \frac{m}{n} \ln 2$$

En que con el análisis visto anteriormente vemos que $O(2k) = O(1)$. Con lo cual de todos modos el costo en memoria usando filtro de Bloom seguirá siendo $O(m + k) = O(n)$, siendo dicho aumento en el costo comparado a no usar el filtro poco significativo, en que además si se compara el costo de almacenar una palabra comparado con el de almacenar un bit, se podría considerar insignificante.

Luego con respecto a los costos temporales de búsqueda, tendremos que sin filtro serán de $O(n)$ correspondientes a buscar en la base cada palabra, mientras que con filtro serán $O(m + n) = O(n)$ dado que primero se busca en la tabla de hash si existe una palabra, y en caso de que el filtro diga que sí, entonces se debe buscar en la base de datos. No obstante, se espera que los tiempos de búsqueda al usar el filtro sean significativamente menores, pues es mucho menos costoso consultar por un bit que recorrer una base de datos completa hasta encontrar una palabra, y se espera que a medida de que la proporción de búsquedas infructuosas aumente, se reduzcan todavía más los tiempos al aplicar el filtro (pues se evitará entrar a la base de datos) y aumenten sin el filtro (pues se recorrerá la base de datos completa fútilmente).

2. Desarrollo

Se realizaron dos archivos *main2.py* y *main.py*, los cuales al ser ejecutados realizan todo el proceso mencionado en esta sección. En estos se encuentran tanto código como los test en función de las variables que serán explicadas en este apartado.

2.1. Limpieza de datos

Como se menciona en la introducción, el filtro de bloom es una interesante herramienta a utilizar cuando se requiere hacer búsquedas en bases de datos extensas, puesto que hace una revisión sobre un arreglo para determinar si ir o no a buscar la información dentro del vasto repositorio de datos.

Para ello hace uso de diferentes funciones de hash, las cuales van mapeando en el arreglo de búsqueda. La problemática aquí es que la construcción de las funciones de hash universales dependen de dos parámetros a y b que están construidos en base al largo máximo de algún elemento de la base de datos a utilizar.

En este caso la base a usar consiste en nombres de bebés contenidos en el archivo *Popular-Baby-Names-Final.csv* en que a su vez para probar búsquedas infructuosas se consideran nombres de películas contenidas en el archivo *Film-Names.csv*. Al estudiar estos archivos se pudo observar que existían nombres de películas extremadamente largos y otros que eran cortos pero que se alargaban porque estaban acompañados de muchos caracteres que no aportaban, por ejemplo 5 puntos y comas (;;;;;) terminando cada string. Entonces se decidió borrar estos últimos caracteres de la base de películas y recortar el largo máximo de la data en 50 para utilizarlo como una constante en el estudio. Asimismo explorando los archivos se observó que habían nombres de películas y bebés que se repetían, con lo cual estos nombres repetidos se eliminaron del archivo de películas.

Esto se realizó mediante el uso de la librería pandas y se puede resumir en las siguientes líneas de código:

```
films = films[films['0'].str.len() <= 50]
films = films[~films['0'].isin(babies['Name'])]
films = films.replace(';;;;;', '', regex=True)
```

2.2. Creación del Filtro de Bloom

2.2.1. Creación de Funciones de Hash

Como se menciona en la sección previa, la construcción del filtro depende de cierta cantidad de funciones de hash, las cuales a su vez dependen de dos parámetros a y b.

Estas funciones de hashing serán de tipo universal y dado el contexto se pueden definir de la siguiente forma:

$$h(x_1, \ldots, x_n) = ((b + \sum_{i=1}^{n} a_i x_i) \mod p) \mod m$$

donde p es un número primo grande a elección y $b \in [0, p-1]$ y los $a_i \in [1, p-1]$ se eligen al azar.

Es por ello que para parámetro a, se creó una función que genera una matriz de k filas y largoMax columnas con valores aleatorios entre 1 y p-1 donde k representa la cantidad de funciones de hash a crear, de manera que cada caracter del string que ingresa como input de la función de hash se va

multiplicando con cada elemento de la fila a[i] sucesivamente para calcular la sumatoria.

La función es la siguiente:

```
#Definiremos Una matriz de tamaño k x max_length con k arrreglos de tamño max_length con valores aleatorios en [1,p-1]
def genera_matriz_a(k, max_length):
    a= np.zeros((k,max_length))
    for i in range(k):
        for j in range(max_length):
            a[i][j]= rd.randint(1, primo-1)

    return a
```

Por otro lado, para b se generó un arreglo de tamaño largoMax con valores entre 0 y p-1 , de manera que la función de hash luego de la sumatoria sumaría el valor correspondiente a la casilla b[i].

La función es la siguiente:

```
#Definiremos un arreglo b de largo k que posee valores aleatorios entre [0,p-1]
def genera_arreglo_b(k):
    b=np.zeros(k)
    for i in range(len(b)):
        b[i]=rd.randint(0,primo-1)

    return b
```

De esta forma, la función de hash que recibe la matriz a y el arreglo b como parámetros, ademas de claramente la palabra y el arreglo M a mapear, se vería de la siguiente forma:

```
def hash(string, a, b, m): #funcion de hash
    ascii_lista = [ord(caracter) for caracter in str(string)]
    x = 0
    for i in range(len(ascii_lista)):
        x += ascii_lista[i]*a[i]
    x += b
    x = (x % primo) % m
    return int(x)
```

2.2.2. Creación de input y Búsqueda

Como ya se mencionó, los datasets utilizados corresponderían a una lista de nombres de bebés y otra de nombres de películas de largos 93890 y 3674 respectivamente, en que la lista de nombre de bebés corresponde a la que se usará para la base de datos. Para crear los test de múltiples búsqueda, se maneja un porcentaje p que indicará qué porcentaje real de los datos otorgados serán de búsquedas exitosas, en que para esto se extraerán p nombres de bebés y el resto serán nombres de películas. Esto según se describe en el siguiente código:

```
def sacar_porcentaje_de_datos(data1,data2, porcentaje,total_datos):
    df1 = pd.read_csv(str(data1))
    df2 = pd.read_csv(str(data2))
```

```
    cantidad_a_sacar = int((total_datos * porcentaje) // 100)
    cantidad_restante = total_datos-cantidad_a_sacar
    filas_aleatorias_df1 = df1.sample(cantidad_a_sacar, replace=True)
    filas_aleatorias_df2 = df2.sample(cantidad_restante, replace=True)
    filas_aleatorias = pd.concat([filas_aleatorias_df1, filas_aleatorias_df2])
    data_final = filas_aleatorias["Name"].to_numpy()
    np.random.shuffle(data_final)
    return(data_final)
```

2.2.3. Estimación de cantidad de funciones de Hash y tamaño del arreglo M

Se puede estimar la cantidad óptima de funciones de hash a utilizar además del tamaño del arreglo M que registra 1's en las posiciones que la función de hash mapea. Esto se puede realizar minimizando o estimando un valor aceptado de falsos positivos.

Asumiendo que el arreglo M tendrá un tamaño m se puede asumir que una función de hash no pondrá un 1 en alguna posición del arreglo con probabilidad $(1-\frac{1}{m})$. Si k es el numero de funciones que utilizaremos y no existe correlación entre ellas se puede decir que la probabilidad que no esté seteado en 1 cualquier posición del arreglo M es $(1-\frac{1}{m})^k$ y en consecuencia como se insertan n elementos, la probabilidad del mismo evento correspondería a $((1-\frac{1}{m})^{kn})$. Finalmente, la probabilidad de un falso positivo sería el complemento de dicha probabilidad pues sería el caso en que en vez de un cero existe un 1, obteniendo una probabilidad de $(1-(1-\frac{1}{m})^{kn})$ para un falso positivo.

Luego notemos que es conocido que $(1-y)^t \approx e^{-yt}$ para un $0 \leq y$ cercano a 0, y veamos que para este caso con una base de datos masiva, se espere que el valor m que defina el tamaño de la tabla de hash sea grande, por tanto $\frac{1}{m}$ es justamente cercano a 0. Por lo tanto el margen probabílistico de falsos positivos que será denominado ϵ es:

$$\epsilon = 1-(1-\frac{1}{m})^{kn} \approx 1-e^{-kn/m}$$

Minimizando esta expresión se obtiene que el valor k que reduce la probabilidad de falso positivo es $k = \frac{m}{n}\ln 2$, lo cual nos permite calcular el valor m igualando ϵ evaluado en k= Kmínimo.

Esto ultimo nos entrega finalmente lo siguiente:

$$\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2}$$

y en consecuencia:

$$k = -\frac{\ln \varepsilon}{\ln 2}.$$

A pesar de ello, la intención de esta actividad corresponde a buscar la cantidad óptima empíricamente, en consecuencia este factor teórico servirá para corroborar la información obtenida.

2.2.4. Testeo

Como se menciona al final del apartado anterior el algoritmo debería ser sometido a diferentes inputs, con la finalidad de estudiar la cantidad óptima de las funciones de hash y el tamaño del arreglo M a utilizar.

Es por esto que se realizaron iteraciones para 6 tamaños de m diferentes y para cada uno de ellos

se aplicaron desde 2 hasta 7 funciones de hash, todo esto en el archivo *main2.py*.

Adicional a esto, para no variar el tamaño del arreglo de búsqueda de búsqueda se utilizó la función sacar-porcentaje-De-datos para los porcentajes 20,40,60 y 80, de manera que la información que se estuviese buscando fuera variando en cada búsqueda, y asimismo ya que dicha función extrae elementos al azar de los dataset en cada iteración se tiene consistencia en los datos extraídos.

El código se presenta a continuación:

```
# create a dataframe after reading .csv file
dataframe = pd.read_csv('Popular-Baby-Names-Final.csv')
t = 1000 #tamaño de los test
n = dataframe.shape[0] #tamaño del csv
maxlen = 50 #solo dejamos los strings hasta con 50 caracteres
primo = 100000019
for m in range(450000, 900001, 90000): #tamaño de m
    for k in range(2, 8): #cantidad de funcions de hash
        p = math.pow(1 - math.exp(-k*n/m), k)
        print("\n" + "m: " + str(m) + " k: " + str(k) + " p: " + str(p) + "\n")
        print()

        M = bitarray.bitarray(m)
        M.setall(0)
        a = genera_matriz_a(k, maxlen)
        b = genera_arreglo_b(k)

        # read csv, and split on "," the line
        csv_file = csv.reader(open('Popular-Baby-Names-Final.csv', "r"), delimiter=",")
        # ingresar valores del csv al filtro
        for row in csv_file:
            nombre = row[0]
            for i in range(k):
                r = hash(nombre, a[i], b[i], m)
                M[r] = 1

        for p in [80, 60, 40, 20]:
            buscar = sacar_porcentaje_de_datos('Popular-Baby-Names-Final.csv', 'Films-Actualizado.
    ↪ csv', p, t)
            c=0 #Número de negativos según el filtro.
            inicio1 = time.time()
            for s in buscar:
                #revisar filtro si tiene que entrar entrar = true
                entrar = True
                for i in range(k):
                    r = hash(s, a[i], b[i], m)
                    if(M[r] == 0):
                        entrar = False
                        c+= 1
                        break
                if entrar:
                    csv_file = csv.reader(open('Popular-Baby-Names-Final.csv', "r"), delimiter=",")
                    for row in csv_file:
                        if s == row[0]:
```

```
44                      break
45              #Si nunca hizo break, el algoritmo dice que lo encontró (podría ser FP)
46          fin1 = time.time()
47          inicio2 = time.time()
48          for s in buscar:
49              csv_file = csv.reader(open('Popular-Baby-Names-Final.csv', "r"), delimiter=",")
50              for row in csv_file:
51                  if s == row[0]:
52                      break #Si hace break, lo encontró
53          fin2 = time.time()
54          print(str(p) + ": Sin filtro demoro " + str(fin2-inicio2) + "s. Con filtro demoro " + str(fin1-
    ↪ inicio1) + \
55              ".\nNúmero de negativos real: " + str(round((1-(p/100))*t)) + ". Número de negativos
    ↪ según el filtro: " + str(c))
```

Luego, una vez se pruebe si hay coincidencia entre los m, k óptimos obtenidos empíricamente y según la teoría, esto mediante la comparación de si la cantidad falsos positivos obtenidos empíricamente coinciden con la teoría, o en caso contrario deben ser mayores para mantener la cantidad de falsos positivos. Luego para probabilidades de error ϵ a escoger se usarán los m, k óptimos empíricos (ya sea coincidan con la teoría o no, pero en caso de que coincidan se obtendrán con los cálculos vistos anteriormente) para continuar los análisis de la comparativa de tiempos entre búsquedas con y sin el filtro de bloom, en función del ϵ y p, todo esto se crea en el archivo *main.py* que cuenta con una implementación casi equivalente a la vista en *main2.py*

3. Resultados

3.1. Equipo Utilizado

Los algoritmos se ejecutaron en un dispositivo con sistema operativo Windows 10, cuyas componentes corresponderían a un procesador intel core i5 de décima generación con 16GB de memoria RAM y tarjeta SSD de 256GB.

3.2. Tablas

Las siguientes tablas muestran los tiempos promedio de una serie de 1.000 búsquedas con distintos valores de m y k. La probabilidad de error representa la probabilidad de un falso positivo en el filtro, se calcula de la siguiente manera: $P_{error} = \epsilon = (1 - e^{-kn/m})^k$ (donde n es la cantidad de elementos ingresados al filtro, en este caso 93.889), el porcentaje de búsquedas exitosas representa cuantos de los elementos del arreglo de búsqueda estaban efectivamente en la base de datos, y el tiempo es el promedio de los 3 resultados que se obtuvieron al hacer los tests.

Tabla 1: Resultados con m = 270.907 y k = 2

Probabilidad de Error	**0,25**			
% de Busquedas Exitosas	**20**		**40**	
	S/ Filtro	**Filtro**	**S/ Filtro**	**Filtro**
Tiempo (s)	25,846436	8,86490417	22,8953769	10,3326428
% de Busquedas Exitosas	**60**		**80**	
	S/ Filtro	**Filtro**	**S/ Filtro**	**Filtro**
Tiempo (s)	20,284852	11,9989309	17,5320835	12,9663122

Tabla 2: Resultados con m = 449.966 y k = 3

Probabilidad de Error	**0,1**			
% de Busquedas Existosas	**20**		**40**	
	S/ Filtro	**Filtro**	**S/ Filtro**	**Filtro**
Tiempo (s)	26,5960426	5,22760558	22,85325	6,96591234
% de Busquedas Existosas	**60**		**80**	
	S/ Filtro	**Filtro**	**S/ Filtro**	**Filtro**
Tiempo (s)	20,3128386	9,5161283	17,5420742	12,6753442

Tabla 3: Resultados con m = 585.419 y k = 4

Probabilidad de Error	**0,05**			
% de Busquedas Exitosas	**20**		**40**	
	S/ Filtro	**Filtro**	**S/ Filtro**	**Filtro**
Tiempo (s)	25,4380686	4,87893748	25,5547078	7,45828271
% de Busquedas Exitosas	**60**		**80**	
	S/ Filtro	**Filtro**	**S/ Filtro**	**Filtro**
Tiempo (s)	23,3801551	10,9613767	19,1384852	13,1154807

Tabla 4: Resultados con m = 899.932 y k = 7

Probabilidad de Error	**0,01**			
% de Busquedas Exitosas	**20**		**40**	
	S/ Filtro	**Filtro**	**S/ Filtro**	**Filtro**
Tiempo (s)	25,6427805	3,16724706	22,9801562	6,66636753
% de Busquedas Exitosas	**60**		**80**	
	S/ Filtro	**Filtro**	**S/ Filtro**	**Filtro**
Tiempo (s)	21,1548662	9,47046423	17,4956765	12,083817

Los resultados muestran que, independientemente de la probabilidad, los resultados sin filtro se mantienen consistentes a lo largo de las tablas; también se puede ver que para los tiempo sin filtro al aumentar el porcentaje de búsquedas exitosas disminuye el tiempo, mientras que para las búsquedas sin filtro el tiempo aumenta.

Esto se debe a que al encontrar el elemento en la base de datos el programa deja de buscar (para disminuir los tiempos de ejecución), debido a esto buscar elementos que no están en la base de datos es más costoso que buscar elementos que si están. Por otro lado las búsquedas con filtro se benefician de un menor porcentaje de búsquedas exitosas ya que esto aumenta la cantidad de veces que puede no ingresar a la base de datos.

3.3. Gráficos

Las gráficas obtenidas a partir de los registros previamente mostrados se presentan a continuación.

3.3.1. Probabilidad de Error con Respecto a M y k

Los siguientes gráficos muestran como varia la probabilidad de un falso positivo con distintos valores de m y k.

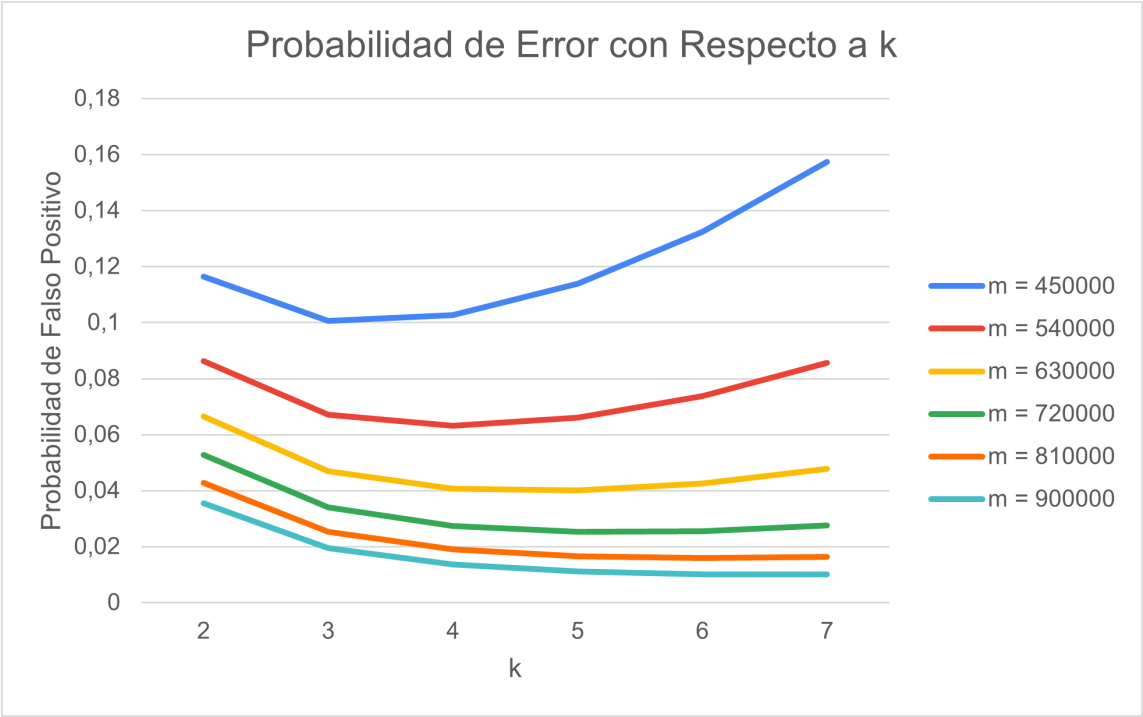

Figura 1: Probabilidad de Error con Respecto a k para Distintos M

Este gráfico condensa toda la información de todos los m y todos los k, para apreciar mejor la existencia de un mínimo el siguiente gráfico muestra una menor cantidad de lineas.

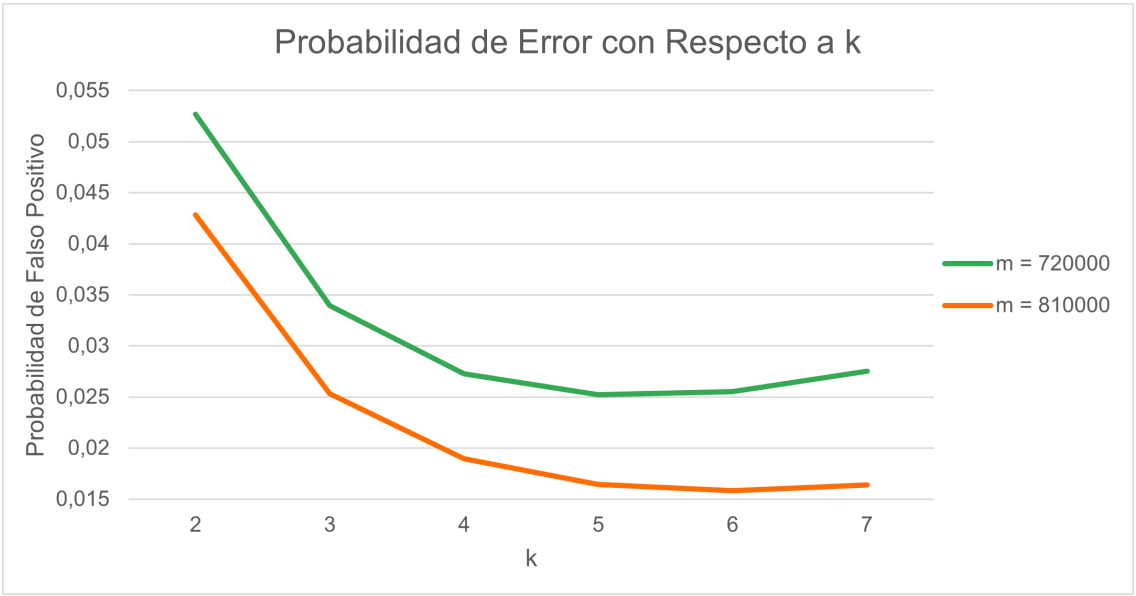

Figura 2: Probabilidad de Error con Respecto a k para Distintos M

3.3.2. Tiempos de Ejecución

Los siguientes gráficos comparan tiempos de ejecución para distintos tipos de búsqueda y distintos arreglos de búsqueda.

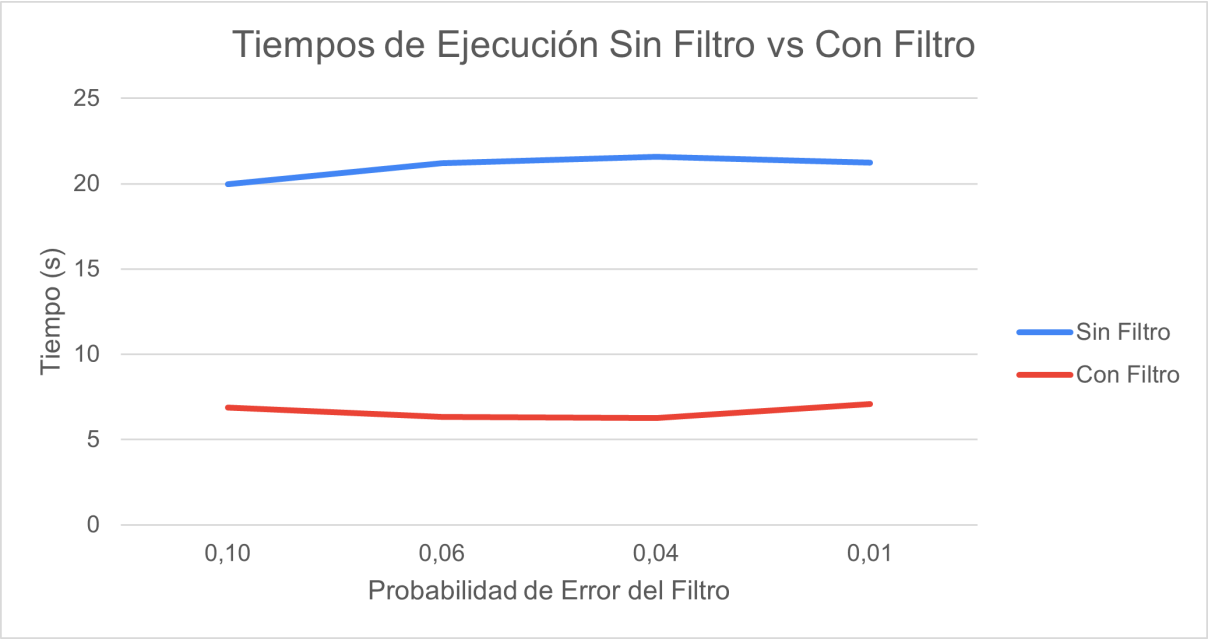

Figura 3: Tiempos de Ejecución con respecto a las probabilidades de error del filtro para un arreglo de búsqueda con un 40 % de búsquedas exitosas

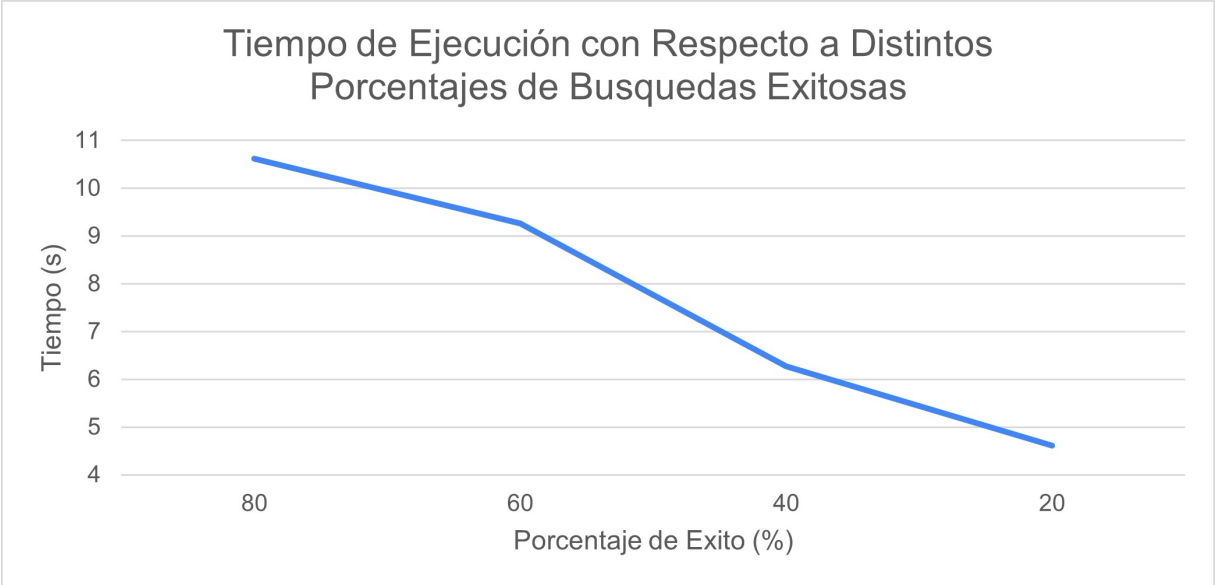

Figura 4: En este gráfico la probabilidad de falso positivo es 4 %

4. Conclusiones

Se puede observar que en función de lo propuesto teóricamente existía una manera muy precisa de poder calcular los valores óptimos del tamaño del arreglo M y de la cantidad de funciones de hash k dadas la probabilidad de error ϵ y el número de inserciones n en la base de datos, los cuales se pudieron corroborar en el apartado anterior, pues los m y k óptimos permitieron obtener la cantidad de falsos positivos esperada, y no se requirió de m y k mayores. Esto se puede apreciar notablemente en que los falsos positivos deberían ser:

$$FP = \epsilon(t(1 - p/100))$$

Esto con los p definidos en el desarrollo. En que este ϵ fue obtenido en *main2.py* según la teoría a partir de m y k, y justamente tiene valores muy similares a los falsos positivos obtenidos empíricamente, que son:

$$FP = \text{Reales Negativos} - \text{Negativos según el filtro}$$

En que luego analizando estos resultados únicamente con los m, k óptimos en *main.py* confirmamos esta relación.

Asimismo dado que los m, k coincidieron con sus valores óptimos, la hipótesis de los costos espaciales y temporales fueron correctas. Sumado a ello, también se comprueba que lo propuesto respecto a los tiempos de búsqueda es correcto; los tiempos obtenidos utilizando el filtro de bloom se redujeron a medida que aumentaba la proporción de datos que no formaban parte de la base de datos inicial, mientras que sin el filtro estos tiempos aumentaban. Esto es coherente tal como se había planteado, que es mucho más rápido y eficiente revisar el arreglo con m bits e ingresar solamente de ser probable que sea necesario, en vez de ingresar de igual forma para los casos de búsqueda infructuosa, teniéndose que llegar a revisar secuencialmente la base masiva de datos completa.

Así se concluye que el filtro de Bloom es una excelente herramienta para las búsquedas extensivas de datos y mucho más útil considerando que los tiempos evolucionan hacia la digitalización y el procesamiento masivo de datos. A su vez este filtro permite reducir los falsos positivos y a su vez los costos temporales si se está dispuesto a ocupar más memoria (aumentando el valor de k y m, en que el mayor costo en memoria claramente corresponde a m) para reducir significativamente el número de falsos positivos y a su vez con esto disminiuir los costos temporales, tal como se pudo apreciar en los resultados obtenidos. En que se puede apreciar además que el costo en memoria extra a pagar no es tan grande puesto que solo se usan bits en el filtro.