Universidad de Costa Rica

Facultad de Ingeniería

Escuela de Ingeniería Eléctrica

IE-0217: Estructuras Abstractas de Datos y Algoritmos para Ingeniería

Proyecto de Investigación:

Bloom Filter

Profesor: Juan Carlos Coto Ulate

Estudiante: Matías Leandro Flores, B94199

II Semestre - 2020

Lunes 14 de diciembre del 2020

Índice

1. Introducción

En el área de la computación, al crear algoritmos, estructuras de datos y otro software para realizar ciertas tareas, el único factor a tomar en consideración no es solamente la eficacia de este. Otros recursos del sistema computacional, como el espacio utilizado, la eficiencia en cuanto al manejo de memoria, y el tiempo requerido para realizar la tarea también son factores importantes, incluso aunque se necesite sacrificar uno por el otro.

Es por esto que en 1970, el científico computacional Burton H. Bloom, en su artículo *Space/Time Trade-offs in Hash Coding with Allowable Erros*, propuso una nueva estructura de datos eficiente en espacio y tiempo [1]. En este trabajo, el autor compara dos métodos de programación por hashing (hash-coding) propuestos con un método tradicional, considerando como factores el espacio de la estructura y el tiempo para identificar un elemento como ausente en dicha estructura. En las palabras de Bloom, "*los nuevos métodos están destinados a* ***reducir la cantidad de espacio necesaria*** *para contener la información asociada con los métodos convencionales. La reducción de espacio es lograda aprovechando la posibilidad de que una* ***pequeña fracción de errores*** *puede ser tolerable*" [2].

De esta manera, surgió una estructura de datos que es eficiente en el espacio utilizado y en el tiempo de ejecución, a cambio de una frecuencia de errores aceptable. Este nuevo método de 'hash-coding' se convirtió en una propuesta útil para aplicaciones en las que la posibilidad de pequeños errores de identificación no es significante, y con el tiempo adoptó el nombre de *'Bloom Filter'*.

2. Discusión

2.1. Funcionamiento

El 'bloom filter' es una estructura de datos probabilística que se utiliza para determinar la membresía de un elemento en un grupo (si el elemento se encuentra o no en el grupo). Esta estructura de datos es muy eficiente, sin embargo existe la posibilidad de obtener 'falsos positivos' al buscar un elemento en el grupo.

Consiste de un vector de **m** bits inicializados en 0, que, según Kumar [3], utiliza una cantidad **k** de funciones de hash para obtener los valores de hash de un elemento. Al ingresar un elemento en el filtro, los bits de los índices del vector de 'm' bits que corresponden a los

‘k’ valores de hash de dicho elemento se “encienden” a un valor de 1. Al buscar un elemento en el filtro, los bits de los índices del vector que corresponden a los ‘k’ valores de hash de dicho elemento se revisan para ver si tienen un valor de 0 o 1. Si uno solo de estos bits es 0, se puede afirmar con certeza que el elemento no se halla en el grupo, sin embargo, si todos los bits son 1 el elemento probablemente está en el grupo.

Los falsos positivos se dan cuando un elemento que probablemente está en el grupo no se encuentra. Luego de insertar *n* elementos al Bloom Filter de tamaño *m* utilizando *k* funciones de hash, la probabilidad de un falso positivo se puede calcular como sigue [4]:

$$P = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{1}$$

Siendo *P* en la ecuación (1) la probabilidad de un falso positivo. Como se puede observar, dado que *P* siempre es un valor menor a 1 elevado a *k*, *P* va disminuyendo mientras mayor es *k* (cantidad de funciones de hash). De acuerdo con Ripeanu y Iamnitchi [4], *P* se minimiza para una cantidad de funciones de hash de:

$$k = \frac{m}{n} ln(2) \tag{2}$$

El Bloom Filter, dado que se utiliza para verificar si un elemento se encuentra o no en un grupo, soporta dos métodos. Un método para insertar elementos al grupo, que, calculando los *k* valores de hash de este elemento, activa los bits de los índices del filtro correspondientes a estos valores de hash a 1. También soporta un método para buscar un elemento en un grupo, el cual, calculando los *k* valores de hash de este elemento, revisa si los bits de los índices del filtro correspondientes a estos valores de hash son 1 o 0. Si todos son 1, el elemento probablemente se encuentra, de lo contrario, definitivamente no se encuentra en el grupo.

Lo más difícil de implementar esta estructura de datos será sin duda generar una cantidad arbitraria de *k* funciones de hash todas diferentes. Para hacer esto, se decidió crear *k* sales de entrada para generar *k* distintas funciones de hash.

Ya habiendo establecido las variables importantes de un Bloom Filter (m, k, n y P) y los métodos que soporta, se elaboró el siguiente pseudocódigo, que pretende ser utilizado de manera general (sin discriminación de lenguaje de programación), para resumir el funcionamiento del Bloom Filter.

Algoritmo 1: Creación de Bloom Filter

Crear constructor: `Argumentos m y k`

```
Atributo de tamaño = m y de cantidad de func. hash = k;
Atributo de cantidad de elementos = 0;
Atributo de sales = array con k 'sales' para k funciones de hash;
Atributo del filtro = array de bits de tamaño m;
Atributo de 'encontrado' inicia en Verdadero;
```

terminar

Método de insertar `Argumento: elemento`

```
Cantidad 'n' de elementos aumenta por 1;
repetir
    Pasar elemento por función hash con la sal[núm. de repetición];
    Convertir la salida de la función hash a entero;
    Valor hash = módulo entre la salida y tamaño 'm' de filtro;
    filtro[valor hash] = 1;
hasta que se cumplan 'k' repeticiones;
```

terminar

Método de buscar `Argumento: elemento`

```
Asumir 'encontrado' es Verdadero;
Calcular probabilidad de falsos positivos con ecuación (1);
repetir
    Pasar elemento por func. hash con sal[núm. de repetición];
    Convertir la salida de la función hash a entero;
    Valor hash = módulo entre la salida y tamaño 'm' de filtro;
    si filtro[valor hash] no es igual a 1 entonces
        'encontrado' es Falso;
    fin
hasta que se cumplan 'k' repeticiones;
Si 'encontrado' es Verdadero elemento probablemente se haya;
```

terminar

2.2. Para qué se utiliza

La utilidad general del *Bloom Filter* es buscar y determinar elementos que no forman parte de un grupo, o en otras palabras, 'filtrar' datos que no forman parte de un set de datos

de información. Por lo tanto, las aplicaciones prácticas de esta estructura de datos están relacionadas a esta utilidad de filtro.

Entre las aplicaciones que ve el *Bloom Filter* está la optimización de los resultados de búsqueda en la web y en correctores ortográficos [5]. Para el primer caso, dado que la mayoría de resultados iniciales de un motor de búsqueda en realidad son muy similares, el *Bloom Filter* sirve para detectar cuáles resultados son probablemente casi idénticos, para así agruparlos o removerlos. Los Bloom Filters son muy útiles en correctores ortográficos ya que permiten determinar si una palabra es válida en un lenguaje, creando un Bloom Filter diccionario que contiene todas las palabras de dicho lenguaje (que como ya se mencionó, no toma mucho espacio ya que es un arreglo de bits) y luego revisando una palabra para determinar si no forma parte de ese diccionario.

El Bloom Filter también ve una aplicación en la filtración de recomendaciones y artículos en Internet [6]. Esto ya que permite filtrar recomendaciones de artículos que no se han leído, y aunque puede excluír algún artículo no leído como si fuera un artículo ya leído (debido a la probabilidad de falsos positivos) eso está bien, ya que el usuario nunca sabe lo que no ve. También se utiliza en Bitcoin para limitar la cantidad de datos de transacción que se reciben, permitiendo solo las transacciones que afectan el monedero del cliente [7], entre varias otras aplicaciones que tiene esta estructura en la filtración de datos.

2.3. Comportamiento en términos de complejidad

Aunque el *Bloom Filter* no es un algoritmo, es posible analizar la complejidad de su tiempo de ejecución, determinando la complejidad de sus operaciones: búsqueda e inserción [8]. El comportamiento en términos de complejidad de las operaciones del *Bloom Filter* se puede determinar mediante la función de acotación de complejidad de máximo, $O(f(n))$, utilizando el modelo RAM.

Para el caso de la **inserción**, observando el pseudo-código de la página 5, se aprecia que el método de insertar consiste de un ciclo que se repite *k* veces, además de la operación de aumentar la cantidad de elementos por 1. La operación de aumentar la cantidad de elementos por 1 constituye un paso sencillo, mientras que el ciclo es una combinación de pasos sencillos ejecutados *k* veces. Por lo tanto:

$$f(n) = 1 + (pasos\ de\ ciclo) \cdot k \tag{3}$$

Dado que la cantidad de pasos del ciclo es constante, la función de máximo $O(f(n))$ para la inserción es entonces:

$$\begin{aligned} O(f(n)) &= O(1 + (pasos\ de\ ciclo) \cdot k) \\ O(f(n)) &= O(1) + (pasos\ de\ ciclo) \cdot O(k) \\ O(f(n)) &\cong (pasos\ de\ ciclo) \cdot O(k) \\ O(f(n)) &\cong O(k) \end{aligned} \tag{4}$$

Esto significa que el tiempo que toma la inserción en un *Bloom Filter* es $O(k)$, dado que solo se debe pasar el elemento por las *k* funciones de hash y agregarlas.

En el caso de la **búsqueda**, esta consiste de un ciclo que es una combinación de pasos sencillos que se repiten *k* veces. También tiene un paso sencillo incial que es asignar *'Verdadero'* a una variable booleana para verificar si se encuentra el elemento. Se tiene por lo tanto lo siguiente:

$$f(n) = 1 + (pasos\ de\ ciclo) \cdot k \tag{5}$$

Por lo tanto la función de máximo $O(f(n))$ para la inserción es:

$$\begin{aligned} O(f(n)) &= O(1 + (pasos\ de\ ciclo) \cdot k) \\ O(f(n)) &= O(1) + (pasos\ de\ ciclo) \cdot O(k) \\ O(f(n)) &\cong (pasos\ de\ ciclo) \cdot O(k) \\ O(f(n)) &\cong O(k) \end{aligned} \tag{6}$$

Como se puede observar, la complejidad del tiempo que toma la búsqueda en un *Bloom Filter* es la misma que para la inserción, $O(k)$. Este valor es extremadamente eficiente, dado que la cantidad *k* de funciones de hash es por lo general una cantidad constante baja.

2.4. Implementación en Python

Al implementar un *Bloom Filter* en Python, sin duda, la mayor dificultad fue crear *k* funciones de hash todas distintas entre sí, siendo *k* cualquier número natural. Para esto, se usó de base la función de hash criptográfica **blake2b**, y para que las *k* funciones de hash sean diferentes se generaron *k* sales de forma aleatoria con el método **os.urandom**, las cuales se utilizaron como un argumento de entrada para la función de hash.

```
1 from bitarray import bitarray
2 from hashlib import blake2b
3 import os
```

```
4
5  class Bloom_Filter:
6      def __init__(self, m, k):
7          self.tamaño = m
8          self.k = k
9          self.n = 0
10         self.filtro = bitarray(m)
11         self.filtro.setall(0)
12         # Array para las 'k' sales que se usarán para varias las
               funcs. de hash
13         self.sales = []
14         for i in range(k):
15             # Crea una sal aleatoria del tamaño de una sal de func.
                   blake2b
16             self.sales.append(os.urandom(blake2b.SALT_SIZE))
17         self.encontrado = True
18
19     def insertar(self, elemento):
20         self.n+=1
21         # Convertir el elemento a forma de bytes que se pueda
               hashear
22         bytelemento = bytes(elemento, 'utf-8')
23         for i in range(self.k):
24             # Aplica función de hash 'blake2b' que varía según su
                   sal
25             func_hash = blake2b(bytelemento, salt = self.sales[i])
26             # Obtiene primeros 5 dígitos de función hash en
                   hexadecimal
27             hexa = func_hash.hexdigest()[:5]
28             # Valor hash = módulo entre func_hash en entero y
                   tamaño de filtro
29             hsh = (hash(hexa) % self.tamaño)
30             # Activa índice de valor hash respectivo: filtro[hsh] =
                   1
31             self.filtro[hsh] = True
32
```

```
33      def buscar(self, elemento):
34          self.encontrado = True
35          # Calcular probabilidad de falso positivo
36          prob = (1-(1-1/self.tamaño)**(self.k*self.n))**self.k
37          bytelemento = bytes(elemento, 'utf-8')
38          for i in range(self.k):
39              # Se procede igual que en insertar() para determinar
                    valor hash
40              func_hash = blake2b(bytelemento, salt = self.sales[i])
41              hexa = func_hash.hexdigest()[:5]
42              hsh = (hash(hexa) % self.tamaño)
43              if self.filtro[hsh] != True:
44                  # Si un índice no está activado, no se encuentra
45                  self.encontrado = False
46          # Si bit es 1 (True) probablemente se encuentra, si no, no
                se encuentra
47          if self.encontrado == True:
48              return "Elemento probablemene se encuentra, con error
                    de %.4f" %(prob)
49          else:
50              return "Elemento definitivamente no se encuentra"
```

Para aplicar la implementación realizada, se elaboró el siguiente problema: Suponiendo que el curso de *Estructuras Abstractas y Algoritmos* es un curso opcional de la carrera de Ing. Eléctrica, se le desea recomendar este curso a estudiantes de la carrera. Para esto, es necesario asegurarse de que los estudiantes a los que se les va a recomendar el curso no lo hayan llevado anteriormente, lo cual no se sabe aún. Se utilizará un *Bloom Filter* para revisar quiénes definitivamente no han llevado el curso, y poder recomendarle el curso a estos estudiantes. Supóngase que los estudiantes que según la base de datos ya llevaron el curso son los estudiantes del II-Ciclo del 2020 (19 estudiantes), registrados por apellido.

Para solucionar este problema se eligió un tamaño *m* del filtro de $m = 90$, que con los $n = 19$ estudiantes registrados, permite una cantidad óptima de $k = 3$ funciones de hash, según la ecuación (2).

```
1 from BloomFilter_B94199 import Bloom_Filter
2
```

```
registrados = ['Alfaro', 'Castrillo', 'Cerdas', 'Corrales', 'Delgado',
               'Gonzales', 'Gutierres', 'Hernandez', 'Hernandez2', 'Herrera',
               'Leandro', 'Mora', 'Muñoz', 'Palacino', 'Poveda', 'Rivel',
               'Sander', 'Stalley', 'Tovar']

recomendar = ['Montealegre', 'Jimenez', 'Troyo', 'Soto', 'Leandro', 'Sander',
              'Corrales', 'Ramirez', 'Rivel', 'Cobain', 'Tovar', 'Palma',
              'Castrillo', 'Rojas','Stalley', 'Alfaro', 'Cubero', 'Palacino',
              'Flores', 'Valdez', 'Herrera', 'Muñoz']

bloom_f = Bloom_Filter(90, 3)
for elemento in registrados:
    bloom_f.insertar(elemento)
# Buscar estudiantes en 'recomendar' para saber si hay alguno que ya llevó el curso
for elemento in recomendar:
    print("{}: {}".format(elemento, bloom_f.buscar(elemento)))
    # Para revisar si es un falso positivo
    if bloom_f.encontrado is True:
        falso = True
        for nombre in registrados:
            if elemento == nombre:
                falso = False
        # Si el elemento encontrado no es igual a ningun elemento en 'registrados', es un falso positivo
        if falso == True:
            print("¡{} es un falso positivo!".format(elemento))
```

La salida del programa ejecutado es la siguiente:

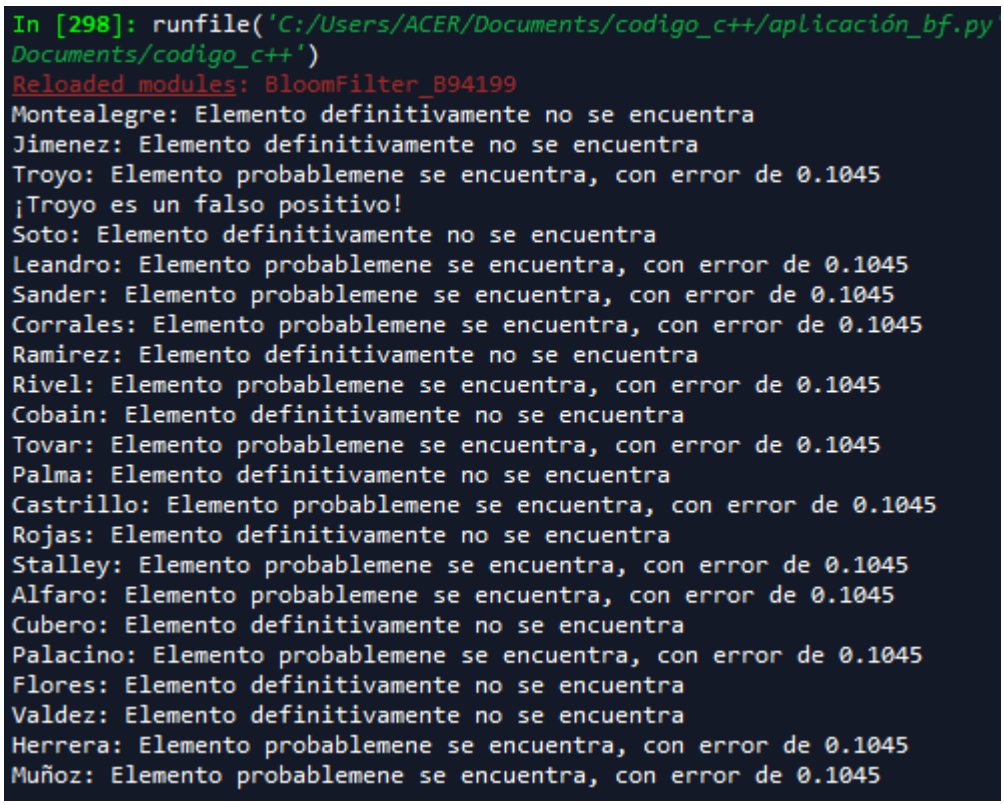

Figura 1: Salida de programa de la implementación del *Bloom Filter*

Como se puede observar, con la implementación desarrollada del *Bloom Filter* se determinó una gran cantidad de estudiantes que no han llevado el curso a los que se les puede recomendar. El único falso positivo obtenido fue en el estudiante “Troyo”, que para un grupo en el que se buscaron 22 elementos, resulta en un error de: $1/22 = 0{,}045$. Esto es un error menor que la probabilidad teórica (1) de un falso positivo, para esta corrida del programa en específico.

3. Conclusiones

El *Bloom Filter*, como se mostró, es una estructura de datos bastante útil y eficiente que ve lugar en muchas aplicaciones. Se explicó el funcionamiento del *Bloom Filter*, el cual probó no ser complicado, al igual que su gran gama de aplicaciones en la filtración de datos e información, y su complejidad, que es bastante baja a cambio de una fracción probabilística de errores. En la implementación del *Bloom Filter* diseñado, que se utilizó para resolver el problema de poder recomendar efectivamente el curso de IE-0217, se utilizó un arreglo de 90 bits, lo cual es una pequeña cantidad de memoria, y para buscar a los estudiantes registrados solo se realizó una cantidad constante de pasos múltiple de $k = 3$, y aún así, con

estas grandes ventajas de eficiencia, solo se obtuvo un falso positivo. Esto significa que el intercambio de una eficiencia en cuanto al tiempo y el espacio por una pequeña fracción de errores es un buen intercambio para la solución de este problema.

En conclusión, el *Bloom Filter* es una estructura abstracta eficiente para solucionar un problema. Su pequeña probabilidad de errores esta bien justificada por su poco espacio y poco tiempo de búsqueda, y es por lo tanto una buena opción para problemas específicos donde el no haber filtrado unos cuantos datos de entre muchos datos no posee significancia.

"I always thought something was fundamentally wrong with the universe"
-**D. Adams**

4. Referencias

[1] K. Nath, “Bloom filter: A simple but interesting data structure.” Medium, https://medium.com/datadriveninvestor/bloom-filter-a-simple-but-interesting-data-structure-37fd53b11606, 2018. (Accesado 7 Dic, 2020).

[2] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[3] A. Kumar, “Bloom filters - introduction and implementation.” https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/, 2017. (Accesado 9 Dic, 2020).

[4] M. Ripeanu and A. Iamnitchi, “Bloom filters – short tutorial.” [Unpublished], 2001.

[5] S. K. Pal and P. Sardana, “Bloom filters their applications,” *International Journal of Computer Applications Technology and Research*, vol. 1, pp. 25–29, 2012.

[6] R. Uda, “Privacy obfuscation with bloom filter for effective advertisement,” in *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pp. 941–946, 2013.

[7] A. Gervais, G. Karame, D. Gruber, and S. Capkun, “On the privacy provisions of bloom filters in lightweight bitcoin clients,” 12 2014.

[8] A. Chumbley, A. Chattopadhyay, A. Sinha, and E. Ross, “Bloom filter.” https://brilliant.org/wiki/bloom-filter/#time-and-space-complexity. (Accesado 7 Dic, 2020).