UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN

Universidad de Concepción

Memoria de título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil Informático

---

Algoritmos paralelos para sketches de cardinalidad y cuentas

Autor:
Estefano Salini Irribarra
Profesores guías:
Cecilia Hernández Rivas
Miguel Figueroa Toro

2 de Enero de 2024
Concepción, Chile

Índice general

Índice de figuras

Índice de cuadros

Resumen

El trabajo exploró el uso de algoritmos de streaming en el contexto de tráficos de red, estimándose la entropía de esta ya que es una medida que sirve para detectar ataques de red. Se implementaron algoritmos de streaming con sketches para la estimación de frecuencias y entropía de trazas de red, con el propósito de mejorar el tiempo de ejecución, el uso de memoria, y/o la precisión en la estimación. Se utilizan sketches ya que los streams de datos son muy grandes, y no es factible guardar en memoria la frecuencia de cada elemento del stream. La solución actual utiliza el sketch CountMin-CU junto a colas de prioridad de tamaño estático (que descartan los elementos menos frecuentes cuando la estructura esta llena y se inserta un nuevo elemento), para la estimación de frecuencias, junto al sketch HyperLogLog, para la estimación de la cardinalidad, para así en conjunto estimar la entropía. Se probó utilizar una modificación que extiende el algoritmo HyperLogLog, permitiendo calcular el histograma de frecuencias de una muestra del stream de datos, para estimar la entropía total y la contribución a la entropía de los elementos descartados por las colas de prioridad asociadas al sketch CountMin-CU. Se propuso también utilizar múltiples sketches CountMin-CU o TowerSketch-CU en conjunto con el sketch HyperLogLog para estimar la entropía. Nuestra implementación del sketch HyperLogLog fue modificada para reducir el uso de memoria en un 33 %. Se encontró que el algoritmo que extiende el sketch HyperLogLog guarda una muestra relativamente representativa del stream de datos, pero que la estimación de la entropía mediante el uso de esta muestra da malos resultados. Se encontró que no hay diferencias en la estimación de la entropía mediante el uso de uno o múltiples sketches CountMin-CU o TowerSketch-CU. Sin embargo, el uso de múltiples sketches CountMin-CU o TowerSketch-CU permite la lectura paralela de las trazas de red mediante OpenMP. Se probó también el algoritmo HyperLogLogLog, una modificación al algoritmo HyperLogLog que reduce el uso de memoria. Se encontró que no reduce el uso de memoria más que nuestra implementación del sketch HyperLogLog. Nuestros resultados indican que implementar el algoritmo utilizando múltiples sketches en una arquitectura con múltiples pipelines (equivalentes a hebras de CPU) reduciría el tiempo de ejecución.

Capítulo 1

Introducción

El tráfico de red de datos corresponde a los paquetes de datos que se transfieren a través de una red de comunicación, donde los paquetes poseen un encabezado con información de la dirección IP fuente, dirección IP destino, puerto, protocolo, y datos a transferir. El conjunto de datos se compone de una secuencia de paquetes, de tamaños definidos de acuerdo al tipo de red, transferidos entre fuente y destino.

La entropía es una de las medidas utilizadas para realizar análisis de tráfico de red y detección de ataques como la negación de servicios. La entropía mide la cantidad de incertidumbre (falta de información) en un evento. Así, la entropía en el tráfico de redes mide la incertidumbre de la cantidad de datos que se mueven en la red. Como la entropía es una medida continua, muchos trabajos usan en su lugar la entropía empírica, la cual se computa usando el número de ocurrencias de las distintas variables.

Debido a que estos flujos de tráfico de red son muy grandes (caben dentro del área de grandes volúmenes de datos) y deben procesarse en tiempo real, procesarlos mediante métodos convencionales no es eficiente, ya que requieren un gran uso de memoria, particularmente teniendo en cuenta que el software que debe llevar a cabo la tarea debe ser implementado en un arquitectura con recursos de memoria limitado (utilizándose el lenguaje de programación P4). Para permitir el procesamiento de muchos datos usando espacios reducidos en memoria existen los algoritmos de streaming y sketches, los cuales permiten computar algunas funciones estadísticas sobre los datos en cualquier momento del procesamiento usando espacio de memoria de trabajo limitado. Un sketch es una estructura de datos que almacena una representación resumida de los datos y que permite responder algunas consultas sobre alguna

característica de los datos. Muchos de estos sketches son aleatorizados por lo que típicamente poseen errores y probabilidades de estimación. Algunos algoritmos permiten estimar cuentas (histograma de frecuencias) y cardinalidad (elementos únicos) de un stream de datos, es decir, de datos que son demasiado grandes para ser almacenados en memoria.

Algunos sketches son HyperLogLog[10], CountMin-CU[1], TowerSketch[8] y ntCard[6]. El sketch HyperLogLog corresponde a una estructura de datos probabilísitca que permite estimar la cardinalidad de un stream de datos, utilizando espacio O(log(log(n))). El algoritmo CountMin-CU corresponde a una estructura de datos probabilística que permite estimar frecuencias de un stream de datos. El algoritmo TowerSketch cumple una funcionalidad similar al algoritmo CountMin-CU. El algoritmo de streaming ntCard sirve para estimar la cardinalidad e histograma de frecuencias de k-mers de un genoma, donde un k-mer corresponde a un substring del genoma. Así, el stream en este caso es el genoma, y los elementos son los k-mer.

En este trabajo se aborda el problema de estimar la entropía empírica de trazas de direcciones IP. Si bien este problema ya tiene una solución[7], se buscó una solución que utilice menos memoria, o bien que sea más rápida o más precisa. Para utilizar menos memoria, se modificó la implementación del sketch HyperLogLog. Para que la solución sea más rápida, se utilizó paralelismo. La solución actual consiste en utilizar la estructura de datos CountMin-CU, para estimar el histograma de frecuencias de las trazas. Dichas frecuencias se almacenan en colas de prioridad, mientras que al mismo tiempo se utiliza la estructura HyperLogLog para estimar la cardinalidad de las trazas. Finalmente, se utilizan estos datos para calcular una estimación de la entropía.

Los algoritmos de streaming pueden hacerse más eficientes mediante el uso de paralelismo, ya sea a nivel de CPU o GPU. Una forma de paralelismo a nivel de CPU es el ofrecido por la plataforma OpenMP, que permite paralelizar código escrito en C/C++ y Fortran.

Una forma de implementar paralelismo en la arquitectura mediante el lenguaje de programación P4, es enviar los paquetes del stream de datos a distintas pipelines, escribiendo el código de modo que no exista dependencia entre las trazas entrantes, de modo que se aproveche el paralelismo de la arquitectura, esto ya que algunas funciones por defecto se ejecutan en paralelo cuando no hay dependencias. En P4 no es posible crear hebras que se ejecuten en paralelo como en otros lenguajes de programación, sin embargo la arquitectura puede poseer múltiples pipelines, donde cada una esta separada de las

otras y procesa paquetes de manera independiente, implementando la misma o diferentes funcionalidades que las demás pipelines, siendo equivalentes a las hebras de CPU[3].

Se propone utilizar una extensión del algoritmo HyperLogLog[4] para soportar el conteo de frecuencias de elementos de una muestra. Se pretende implementar el algoritmo tal como ya está implementado en el paper, ya que el algoritmo en sí está contando la frecuencia de cada dato demográfico (en este caso las direcciones IP), lo cual puede traducirse como estimar las cuentas de los datos demográficos. Además, se pretenden utilizar los resultados para calcular la entropía de la muestra, para así realizar todo el procedimiento dado por las estructuras CountMin-CU, colas de prioridad y HyperLogLog, utilizando solo una estructura HyperLogLog. Se deben comparar los resultados y sus errores de estimación con los valores dados por la solución ya existente.

Este trabajo también explora implementar el algoritmo del estado del arte[7] de una manera que permita mayor paralelismo y con otros sketches de cuentas mas recientes como es el TowerSketch y TowerSketch-CU. De la misma manera se desea implementar y evaluar el sketch HyperLogLogLog[9] en lugar del algoritmo HyperLogLog para la estimación de elementos distintos.

Capítulo 2

Objetivos

2.1. Objetivo general

El objetivo general es implementar y evaluar algoritmos de streaming basados en sketches para calcular la cardinalidad y estimar cuentas usando paralelismo en OpenMP.

2.2. Objetivos específicos

1. Implementar y evaluar la extensión del algoritmo HyperLogLog para estimar cuentas bajo el contexto de trazas de direcciones IP.

2. Implementar y evaluar el algoritmo HyperLogLogLog para estimar cardinalidad.

3. Implementar y evaluar el algoritmo HyperLogLog junto a múltiples sketches CountMin-CU para estimar cuentas bajo el contexto de trazas de direcciones IP.

4. Implementar y evaluar el algoritmo HyperLogLog junto al sketch Tower-Sketch-CU para estimar cuentas bajo el contexto de trazas de direcciones IP.

5. Diseñar e implementar algoritmos para estimar cardinalidad y cuentas de forma paralela.

6. Evaluar el espacio y tiempo de ejecución y estimación de sketches de los algoritmos.

Capítulo 3

Marco Teórico

3.1. Algoritmos de streaming y sketches

Un algoritmo de streaming corresponde a un algoritmo que recibe como entrada un stream de datos, es decir, una secuencia de elementos, y que puede ser examinado pocas veces, usualmente solo una vez. Específicamente, es una estructura de datos dinámica donde la estructura de datos debe utilizar memoria menor a o(b), donde b son los bits de memoria que representan a los datos, ya que hay tantos elementos que solo se pueden almacenar una fracción de ellos. El objetivo es tener como salida un resumen o sketch de los datos con el que se puedan responder preguntas acerca de los datos.

El sketch es una estructura de datos probabilística (debido al uso de funciones de hashing) que ocupa espacio reducido, típicamente sublineal, la cual permite resolver una o más consultas sobre alguna característica de los datos.

3.1.1. Estimación de elementos distintos

Los algoritmos de estimación de elementos distintos corresponden a algoritmos de streaming cuyo sketch es utilizado para calcular la cardinalidad de un conjunto de datos, donde la cardinalidad corresponde a la cantidad de elementos distintos en ese conjunto de datos. Entre los algoritmos de streaming para la estimación de elementos distintos se encuentran los algoritmos HyperLogLog y su variante HyperLogLogLog, y el algoritmo ntCard.

HyperLogLog

Corresponde a un algoritmo de streaming para estimar la cardinalidad de un conjunto de datos. Posee una estructura de datos sketch que corresponde a un vector de un largo definido por el error que se desee en el cálculo de la cardinalidad, donde el valor de cada celda está inicializada en cero. A cada elemento que se lee en el stream se le calcula una función de hash, a la cual se divide su representación binaria en dos partes: la primera, de largo p (donde p define el tamaño del sketch, 2^p), indica el indice de la celda en el sketch, y la segunda, de largo b (donde la suma de p y b es igual al largo de la funcion de hash), se utiliza para calcular un "registro" que posteriormente será almacenado en la celda correspondiente. Este registro se calcula mediante el cálculo de la posición del primer 1, de izquierda a derecha, de la parte derecha de la representación binaria de la función de hash calculada. El valor de esta posición indica qué tan probable es que se encuentre el elemento en el stream, donde un valor mayor indica que el elemento en el stream es menos probable. Si el valor del registro calculado es mayor al almacenado en la celda correspondiente, se reemplaza el valor guardado en la celda por el valor del registro calculado. Una vez que se ha procesado todo el stream, se procede a calcular la media armónica del sketch mediante la siguiente fórmula:

$$E_{arm} = \sum \frac{1}{2^w} \tag{3.1}$$

La razón de reemplazar los registros de menor valor por los de mayor valor, es decir, los registros de elementos más probables por lo menos probables, es que un elemento menos probable indica que ya se pasó múltiples veces por un elemento más probable.

La figura 3.1 muestra una representación del algoritmo, donde se está insertando un registro de valor 4 en la celda con índice 3, mientras que el algoritmo 1 muestra el algoritmo de inserción al sketch HyperLogLog.

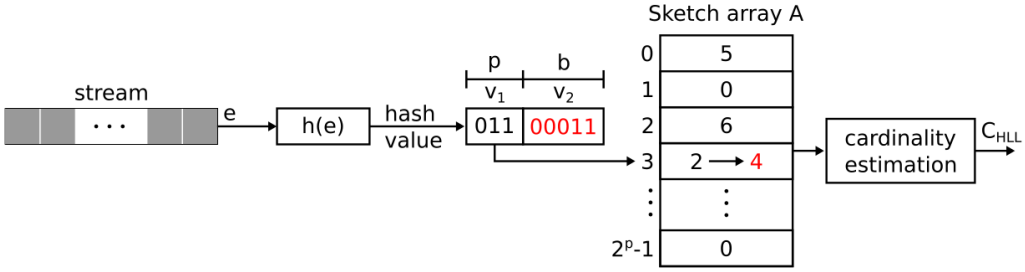


Figura 3.1: Representación del algoritmo HyperLogLog. Obtenido de A High-Throughput Hardware Accelerator for Network Entropy Estimation Using Sketches[7]

Algoritmo 1 Inserción a HyperLogLog

1: **Input** y
2: **Output** M
3: $j \leftarrow f(y)$
4: $r \leftarrow \rho(h(y))$
5: $r_0 \leftarrow M[j]$
6: **if** $r > r_0$ **then**
7: $\quad M[j] \leftarrow r$
8: **end if**

Una vez calculada la media armónica, se procede a calcular la estimación de la cardinalidad mediante la siguiente fórmula:

$$C_{HLL} = \alpha_m \cdot N^2 \cdot (\sum \frac{1}{2^w})^{-1} \tag{3.2}$$

Donde N es el tamaño del sketch, w es el valor del registro, y el valor de α_m es igual a 0.673 para $N = 16$; es igual a 0.697 para $N = 32$; es igual a 0.709 para $N = 64$, y si $N \geq 128$ entonces está determinado mediante la siguiente fórmula:

$$\alpha_m = \frac{0{,}7213}{1 + \frac{1{,}079}{N}} \tag{3.3}$$

Una vez calculada la estimación de la cardinalidad, se procede, de ser necesario, a calcular una corrección de la cardinalidad estimada. Esto solo se hace si se trabajan con funciones hash de 32 bits, lo cual corresponde al caso

nuestro. Si la cardinalidad estimada es menor o igual a $5 \cdot \frac{N}{2}$, entonces hay demasiados ceros, y se estima la cardinalidad con la siguiente fórmula:

$$C_{HLL} = N \cdot \log \frac{N}{0^s} \tag{3.4}$$

Por otro lado, si la cardinalidad estimada es mayor a $\frac{2^{32}}{30}$, la estimación es demasiado alta, y se corrige mediante la siguiente fórmula:

$$C_{HLL} = -2^{32} \cdot \log(1 - \frac{C_{HLL}}{2^{32}}) \tag{3.5}$$

El algoritmo de estimación de la cardinalidad se presenta a continuación

Algoritmo 2 Estimación de cardinalidad

Input M, N
Output C_{HLL}
$C_{HLL} \leftarrow 0$
for $m \in M$ **do**
    $C_{HLL} \leftarrow C_{HLL} + \frac{1}{2^m}$
end for
$\alpha_m \leftarrow \frac{0{,}7213}{1+\frac{1{,}079}{N}}$
$C_{HLL} \leftarrow \frac{\alpha_m \cdot N^2}{C_{HLL}}$
if $C_{HLL} \leq 5 \cdot \frac{N}{2}$ **then**
    $card \leftarrow N \cdot \log \frac{N}{zeros}$
else if $C_{HLL} > \frac{2^{32}}{30}$ **then**
    $C_{HLL} \leftarrow -2^{32} \cdot \log(1 - \frac{C_{HLL}}{2^{32}})$
end if

El algoritmo utiliza espacio O(log(log(n))), donde el primer logaritmo corresponde al registro que tiene como valor máximo la cantidad de bits de la función de hash, es decir, al logaritmo del valor máximo de la función de hash, y el segundo logaritmo corresponde al tamaño del sketch, el cual es a lo más el logaritmo del tamaño de la entrada. Además, se ejecuta en tiempo $O(n + 2^p)$, donde n es el largo del stream, y 2^p es el tamaño del sketch.

Es posible realizar la unión de dos sketches HyperLogLog. Para ello creamos un sketch nuevo, con las celdas inicializadas en 0, y recorremos al mismo tiempo los dos sketches que serán unidos. Por cada celda con un mismo índice

de los sketches, nos quedamos con el registro mayor, el cual lo insertamos al nuevo sketch en la celda con el índice correspondiente.

Algoritmo 3 Unión de sketches HyperLoglog

```
1: Input M_1, M_2, N
2: Output M
3: M ← sketch de tamaño N inicializado en ceros
4: for i ← 0 to N do
5:     m_1 ← M_1[i]
6:     m_2 ← M_2[i]
7:     if m_1 > m_2 then
8:         M[i] ← m_1
9:     else
10:        M[i] ← m_2
11:    end if
12: end for
```

Es posible utilizar el algoritmo HyperLogLog pero con uso de memoria reducido. Para ello, se utiliza la variante HyperLogLogLog.

HyperLogLogLog

El algoritmo HyperLogLogLog es similar al algoritmo HyperLogLog, pero utiliza, en vez de un solo sketch, un sketch comprimido y un mapa, implementado como tabla hash, con los elementos que no están en el sketch comprimido. El sketch comprimido contiene registros que están dentro de una región densa que se define como los registros que son mayores o iguales a una variable llamada base, que representa un valor de un registro (inicializado en 0), y menores a la suma de la base con 2^κ, donde κ es un parámetro que define cuantos bits debe tener cada registro dentro del sketch comprimido. Para la implementación de esta memoria, se definió un κ de valor 3, es decir, los registros almacenados dentro del sketch comprimido utilizan 3 bits. Esto es posible si antes de almacenar un registro en el sketch, se le resta a este la base. Los registros que están fuera de la región densa se almacenan en el mapa.

Cada vez que se lee un elemento del stream, se debe llamar a la operación UPDATE. En el algoritmo UPDATE, primero obtenemos el índice a partir

de la mitad izquierda de la función hash, luego obtenemos el registro a partir de la posición del primer 1 de izquierda a derecha de la mitad derecha de la función hash. Posteriormente obtenemos el registro almacenado en el sketch comprimido o mapa, llamando para ello a la función GET, y comparamos dicho registro con el obtenido a partir de la función hash. Si el registro calculado es mayor al almacenado, entonces debemos revisar si el registro está dentro de la región densa. De estarlo, se le resta al registro la base y se almacena en el sketch comprimido (M), en el índice correspondiente. Además, se debe eliminar el registro almacenado del mapa (S), de estar en este. Si el registro calculado es mayor al previamente almacenado y además el registro calculado no está en la región densa, se inserta en el mapa. Luego, si llegamos a insertar el registro en alguna estructura, llamamos a la operación COMPRESS.

Algoritmo 4 UPDATE

```
1: Input S, M, B, y
2: Output S, M
3: j ← f(y)
4: r ← ρ(h(y))
5: r_0 ←GET(S, M, B, j)
6: if r > r_0 then
7:     if B ≤ r < B + 2^κ then
8:         M[j] ← r − B
9:         if j ∈ S then
10:            del S[j]
11:        end if
12:    else
13:        S[j] ← r
14:    end if
15:    COMPRESS(S, M, B)
16: end if
```

El algoritmo GET primero busca si la clave está en el mapa S utilizando el índice j como clave, de lo contrario lo busca en la celda de índice j en el sketch comprimido M. Ya que en este último caso se obtiene un elemento de 3 bits, se le debe sumar la base B para obtener el registro completo.

El algoritmo COMPRESS consiste en primero encontrar el valor de la base que maximice la cantidad de registros almacenados que entren en la

Algoritmo 5 GET

1: **Input** S, M, B, j
2: **Output** r_0
3: **if** $j \in S$ **then**
4: $\quad r_0 \leftarrow S[j]$
5: **else**
6: $\quad r_0 \leftarrow B + M[j]$
7: **end if**

región densa. Si el valor encontrado es diferente al valor actual de la base, se debe llamar al algoritmo REBASE.

Algoritmo 6 COMPRESS

1: **Input** S, M, B
2: **Output** S, M, B
3: $B' \leftarrow \text{argmax}_{B'' \in [w]} |\{j : B'' \leq \text{GET}(S, M, B, j) < B'' + 2^\kappa\}|$
4: **if** $B' \neq B$ **then**
5: $\quad \text{REBASE}(S, M, B, B')$
6: **end if**

El algoritmo REBASE consiste en revisar, por cada valor del índice (es decir, desde 0 hasta $2^p - 1$), si el registro correspondiente está dentro de la región densa, para luego revisar si este debe ir en el sketch comprimido o en el mapa.

Para hacer la unión de dos sketches, se utiliza el algoritmo MERGE, el cual consiste en primero crear un nuevo mapa vacío y un sketch comprimido inicializado con ceros. Luego obtenemos el valor de la base de este sketch como el valor máximo entre las dos bases de los sketches a unir. Luego, por cada valor del índice, obtenemos los registros de los sketches a unir, y nos quedamos con el mayor. Si este registro está dentro de la región densa dada por la base, se inserta en el sketch comprimido, de lo contrario se inserta en el mapa. Finalmente, llamamos al algoritmo COMPRESS.

Para mejorar el tiempo de ejecución, se utiliza una heurística del paper del algoritmo HyperLogLogLog que se describe a continuación: se llama a COMPRESS solo si se inserta un nuevo elemento al mapa S, se prueba solo el siguiente valor posible de registro que es mayor a B y se omiten las otras opciones.

Algoritmo 7 REBASE

Input S, M, B, B'
Output S, M, B
for $j \in [m]$ **do**
    $r \leftarrow$GET(S, M, B, j)
    if $B' \leq r < B' + 2^\kappa$ **then**
        $M[j] \leftarrow r - B'$
        if $j \in S$ **then**
            del $S[j]$
        end if
    else
        $S[j] \leftarrow r$
    end if
end for
$B \leftarrow B'$

Algoritmo 8 MERGE

Input $S_1, M_1, B_1, S_2, M_2, B_2$
Output S, M, B
$S \leftarrow \phi$
Inicializar $M \in [2^\kappa]^m$ con ceros
$B \leftarrow$max$\{B_1, B_2\}$
for $j \in [m]$ **do**
    $r_1 \leftarrow$GET(S_1, M_1, B_1)
    $r_2 \leftarrow$GET(S_2, M_2, B_2)
    $r \leftarrow$max$\{r_1, r_2\}$
    if $B \leq r < B + 2^\kappa$ **then**
        $M[j] \leftarrow r - B$
    else
        $S[j] \leftarrow r$
    end if
end for
COMPRESS(S, M, B)

ntCard

Corresponde a un algoritmo de streaming para estimar la cardinalidad de un conjunto, así como la cantidad de elementos con frecuencia 1 hasta frecuencia t_{max}. El algoritmo utiliza como estructuras de datos a arreglos de largos 2^r o t_{max}. Hace uso de una función de hash de 64 bits llamada ntHash, la cual posee garantías especiales. Se utilizan específicamente los primeros s bits y los últimos r bits. Si los primeros s bits son iguales a 0, entonces se toman los primeros r bits de la función de hash, los cuales indicarán el índice de un arreglo de largo 2^r llamado tabla de multiplicidad, la cual se utiliza posteriormente para estimar la cardinalidad, y la cantidad de elementos con frecuencia 1 hasta frecuencia t_{max}, donde t_{max} es el valor máximo en la tabla de multiplicidad. El pseudocódigo del algoritmo se muestra en los algoritmos 9 y 10.

Algoritmo 9 Update

```
1: Input Stream de datos
2: Output t
3: for cada secuencia leida do
4:     h ←ntHash(secuencia)
5:     if h_{64:64−s+1} = 0^s then
6:         i ← h_{r:1}
7:         t_i ← t_i + 1
8:     end if
9: end for
```

No es factible utilizar el algoritmo ntCard para estimar el histograma de frecuencias de trazas de red, esto debido a que el algoritmo ntCard está diseñado para trabajar con la función de hash ntHash, y dicha función de hash a su vez está diseñada para trabajar con bases de genomas.

3.1.2. Estimación de elementos frecuentes

Los algoritmos de estimación de elementos frecuentes corresponden a algoritmos de streaming cuyos sketches contienen una estimación de las frecuencias de los elementos de un conjunto de datos. Con estos algoritmos es posible construir el histograma de frecuencias del conjunto de datos. Entre los algoritmos de streaming para la estimación de elementos frecuentes se

Algoritmo 10 Estimate

Input t
Output F_0,f
for $i \leftarrow 1$ to 2^r **do**
    $p_{t[i]} \leftarrow p_{t[i]} + 1$
end for
for $i \leftarrow 1$ to t_{max} **do**
    $p_i \leftarrow p_i / 2^r$
end for
$F_0 \leftarrow -\ln p_0 \cdot 2^{s+r}$
for $i \leftarrow 1$ to t_{max} **do**
    $f'_i \leftarrow \frac{-p_i}{p_0 \ln p_0} - \frac{1}{i} \sum_{j=1}^{i-1} \frac{j \cdot p_{i-j} f'_j}{p_0}$
end for
for $i \leftarrow 1$ to t_{max} **do**
    $f_i \leftarrow f'_i \cdot F_0$
end for

encuentran los algoritmos CountMin y su variante CountMin-CU, y el algoritmo TowerSketch.

CountMin-CU

Corresponde a un algoritmo de streaming para estimar las frecuencias de los elementos más frecuentes en un stream de datos. Consiste en una matriz de d filas y w columnas, donde un elemento llega a una fila d veces y se le aplica una función de hash, es decir, en total se le aplican d funciones hash. Para estimar la frecuencia de un elemento, se toma el valor mínimo de los valores guardados en las d filas a las que se hashea el elemento. El algoritmo permite las operaciones de inserción y de estimación de frecuencia de un elemento. Para ambos casos, es necesario entregar el elemento, ya que el sketch no lo almacena. El algoritmo nunca subestima las frecuencias, pero si las puede sobrestimar. Se diferencia de su predecesor, CountMin[5], en que en la versión CU al actualizarse el contador de las celdas, se incrementan solo los contadores que en ese momento poseen la frecuencia estimada, es decir, cuyo valor almacenado corresponde al valor de la frecuencia del elemento insertado, calculado como se indicó anteriormente. En el algoritmo original, se actualiza el contador aunque su valor sea diferente al de la frecuencia

estimada actual del elemento insertado. Utiliza espacio $O(d \cdot w \cdot b)$, donde b son los bits utilizados por cada celda de la matriz.

Algoritmo 11 Inserción a CountMin-CU

```
1: Input key, d
2: minval ← 2^32 − 1
3: for i ← 0 to d do
4:     hashval ← hash(key, i) %w
5:     if minval ≥ C[i][hashval] then
6:         minval ← C[i][hashval]
7:     end if
8: end for
9: for i ← 0 to d do
10:    hashval ← hash(key, i) %w
11:    if minval = C[i][hashval] then
12:        C[i][hashval] ← C[i][hashval] + 1
13:    end if
14: end for
```

Algoritmo 12 Estimación de contador CountMin-CU

```
1: Input key
2: Output minval
3: minval ← 2^32 − 1
4: for i ← 0 to d do
5:     hashval ← hash(key, i) %w
6:     if minval ≥ C[i][hashval] then
7:         minval ← C[i][hashval]
8:     end if
9: end for
```

TowerSketch

Al igual que CountMin-CU, corresponde a un algoritmo de streaming para estimar las frecuencias de los elementos más frecuentes en un stream de datos. Consiste en arreglos de contadores y d funciones hash, donde d por defecto es igual a 5. El algoritmo utiliza contadores de diferentes tamaños

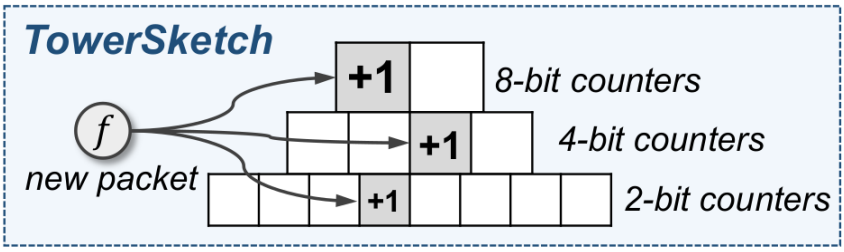


Figura 3.2: Representación del sketch TowerSketch. Obtenido de SketchINT: Empowering INT with TowerSketch for Per-flow Per-switch Measurement[8]

para los distintos arreglos, pero utiliza la misma cantidad de memoria en cada arreglo. TowerSketch organiza los contadores en una forma de torre con d arreglos. Por cada dos arreglos adyacentes, el arreglo en un nivel superior tiene menos contadores, los cuales son de mayor tamaño, de modo que la cantidad de bits utilizados por arreglo es la misma. El algoritmo permite la inserción y estimación de un elemento. El algoritmo tiende a sobrestimar los contadores para streams de datos grandes. TowerSketch tiene la variante TowerSketch-CU, donde, al igual que en el algoritmo CountMin-CU, solo se incrementa el contador más pequeño (y donde no se produzca overflow).

La figura 3.2 muestra una representación del sketch TowerSketch, donde los tres arreglos utilizan la misma cantidad de memoria, pero los contadores son de diferente tamaño, con los arreglos superiores teniendo menos contadores pero de mayor tamaño. El algoritmo de inserción al sketch TowerSketch se muestra en el algoritmo 13, y el algoritmo de consulta se muestra en el algoritmo 14.

El algoritmo de inserción recibe como input el elemento a insertar (key), el largo (en bytes) del elemento, un arreglo w que indica la cantidad de contadores en un arreglo del sketch, un arreglo de semillas para la función hash MurmurHash3, y el sketch TowerSketch (A). Definimos d en 5 y los arreglos de constantes de largo d: cs, cpw, lo, y $mask$. El arreglo cs contiene offsets que se aplicarán a la variable $shift$. El arreglo cpw contienen offsets que se utilizan para determinar el contador correspondiente en el arreglo A_i. El arreglo lo corresponde a una máscara de bits, donde los arreglos inferiores tienen una máscara más larga. El arreglo $mask$ contiene máscaras de bits que se aplicarán a la variable val, donde los arreglos superiores tienen máscaras más largas, ya que utilizan más bits por contador. La variable a corresponde a un puntero hacia el contador correspondiente. El algoritmo recorre la torre de

Algoritmo 13 Inserción al sketch TowerSketch

1: **Input** $key, key\_len, w, hashseed, A$
2: $cs \leftarrow \langle 1, 2, 3, 4, 5 \rangle$
3: $cpw \leftarrow \langle 4, 3, 2, 1, 0 \rangle$
4: $lo \leftarrow \langle 0xf, 0x7, 0x3, 0x1, 0x0 \rangle$
5: $mask \leftarrow \langle 0x3, 0xf, 0xff, 0xffff, 0xffffffff \rangle$
6: $idx \leftarrow \langle 0, 0, 0, 0, 0 \rangle$
7: $d \leftarrow 5$
8: **for** $i \leftarrow 0$ to d **do**
9: $idx[i] \leftarrow$MurmurHash3$(key, key_len, hashseed[i])\ \%w[i]$
10: $\&a \leftarrow A[i][idx[i] \gg cpw[i]]$
11: $shift \leftarrow (idx[i] \& lo[i]) \ll cs[i]$
12: $val \leftarrow (a \gg shift) \& mask[i]$
13: **if** $val < mask[i]$ **then**
14: $a \leftarrow a + (1 \ll shift)$
15: **end if**
16: **end for**

abajo hacia arriba, donde cada fila ocupa $32 \cdot w\_d$ bits, ya que son arreglos de enteros sin signo de 32 bits. La fila de primer nivel tiene $w\_d \cdot 2^4$ elementos, mientras que la fila de nivel más alto tiene w_d elementos. Esto significa que los niveles inferiores tienen múltiples elementos por celda, por lo que al aumentar el contador de una celda, se aumenta el contador de múltiples elementos. Por eso, se aumentan los contadores de las celdas de las filas inferiores con menor frecuencia y en valores mayores o iguales a 1. Notemos que el valor val tendrá un valor máximo de $0x3$ para la primera fila, y de $0xffffffff$ para la última fila, debido a la máscara $mask$ que se aplica. Esto significa que el condicional de la línea 13 es más probable que se cumpla para la última fila que para las filas inferiores. Además, notemos que el valor shift tendrá un valor máximo de 30 para la primera fila, y de 0 para la última fila, debido a la máscara lo que se aplica, y al desplazamiento de cs bits a la izquierda. Así, en la línea 14, se aumentará el contador de una celda de la primera fila en un valor entre 1 y 2^{30}, y para una celda de la última fila será en un valor de 1.

El algoritmo de consulta es similar al algoritmo de inserción, pero no modifica los contadores, y en su lugar solo busca el valor mínimo que corresponde a la frecuencia del elemento que se está consultando. Notemos que el

Algoritmo 14 Consulta al sketch TowerSketch

Input $key, key\_len, w, hashseed, A$
Output ret
$cs \leftarrow \langle 1, 2, 3, 4, 5 \rangle$
$cpw \leftarrow \langle 4, 3, 2, 1, 0 \rangle$
$lo \leftarrow \langle 0xf, 0x7, 0x3, 0x1, 0x0 \rangle$
$mask \leftarrow \langle 0x3, 0xf, 0xff, 0xffff, 0xffffffff \rangle$
$idx \leftarrow \langle 0, 0, 0, 0, 0 \rangle$
$d \leftarrow 5$
$ret \leftarrow 2^{32} - 1$
for $i \leftarrow 0$ to d **do**
    $idx[i] \leftarrow$MurmurHash3$(key, key_len, hashseed[i])\ \%w[i]$
    $\&a \leftarrow A[i][idx[i] \gg cpw[i]]$
    $shift \leftarrow (idx[i] \& lo[i]) \ll cs[i]$
    $val \leftarrow (a \gg shift) \& mask[i]$
    if $val < mask[i]$ and $val < ret$ **then**
        $ret \leftarrow val$
    end if
end for

algoritmo devuelve el valor *val*, y que a este se le aplica la máscara *mask*, donde su valor es menor para las filas inferiores. Esto porque los contadores de las filas inferiores son de menor tamaño, donde los contadores de la primera fila (la más inferior) son de 2 bits, mientras que los de la última fila (la más superior) son de 32 bits.

3.2. Estimación de elementos frecuentes con sampling y sketches

Es posible utilizar el algoritmo HyperLogLog para estimar las cuentas de una muestra[4]. Para ello, cada celda del sketch contendrá ahora 4 elementos. El primero es el registro. El segundo es un indicador auxiliar, el cual se determina con los bits de la parte derecha de la función de hash. El tercero es el contador de frecuencias del elemento de la muestra. El cuarto y último es la muestra demográfica que consiste en el elemento de la muestra almacenado en dicha celda del sketch, en este caso una dirección IP.

El algoritmo de inserción de un elemento del stream al sketch se muestra en el algoritmo 15, donde la línea 13 indica que se encuentra con un elemento menos probable que el actual, ya sea por un registro mayor o por un indicador auxiliar mayor. En este caso, se reinicia el contador de frecuencias y se actualiza la información demográfica almacenada por la recién vista. Por otro lado, la línea 17 indica que nos encontramos con un mismo elemento, o bien hay una colisión en la función de hash. Se aumenta en 1 el contador de frecuencias. En caso de una colisión, nos quedamos con solo una de las informaciones demográficas, de acuerdo a un criterio pre-definido (el cual no fue indicado en el paper y queda a elección de la implementación del algoritmo).

3.3. Uso de sketches para la estimación de entropía

De acuerdo a la teoría de la información, la entropía corresponde a una medida de la incertidumbre asociada a una variable aleatoria. En el contexto de trazas de red, corresponde a la aleatoriedad recolectada por una aplicación mediante hardware (un router específicamente) para usos en criptografía u otros usos que requieran datos.

Algoritmo 15 Insert

Input stream,p,b
Output M, I, F, D
$N \leftarrow 2^p$
Inicializar N registros $M[0], ..., M[N-1]$ en 0
Inicializar N indicadores auxiliares $I[0], ..., I[N-1]$ en 0
Inicializar N contadores de frecuencia $F[0], ..., F[N-1]$ en 0
Inicializar N muestras demográficas $D[0], ..., D[N-1]$ en *null*
for $e \in$ stream **do**
    $x \leftarrow h(e)$
    $j \leftarrow \langle x_0, ..., x_{p-1} \rangle_2$
    $i \leftarrow \langle x_p, ..., x_{p+b-1} \rangle_2$
    $w \leftarrow (\min\{t | t \geq 0 \text{ and } t < b, x_{p+t} = 1\}) + 1$
    if $M[j] < w$ or ($M[j] = w$ and $I[j] < i$) **then**
        $I[j] \leftarrow i$
        $F[j] \leftarrow 1$
        $D[j] \leftarrow e$
    else if $M[j] = w$ and $I[j] = i$ **then**
        $F[j] \leftarrow F[j] + 1$
        if $D[j] \neq e$ **then**
            Se escoge $D[j]$ de $(D[j], e)$ a través de un criterio pre-definido
        end if
    end if
    $M[j] \leftarrow \max(M[j], w)$
end for

De acuerdo al estado del arte[7], se utilizan los algoritmos HyperLogLog y CountMin-CU en conjunto para estimar la entropía. El sketch CountMin-CU estima la frecuencia de los elementos más frecuentes, los cuales corresponden a los elementos dentro de un conjunto de datos cuyas frecuencias están dentro de las más repetidas. En un histograma de frecuencias, corresponden a los datos que están en el peak de una curva sesgada a la derecha. Por otro lado, el sketch HyperLogLog estima la cardinalidad de los elementos entrantes, valor que es utilizado en la fórmula para calcular la entropía. Se utiliza la estructura CountMin-CU con parámetros d=8 y w=16384, la cual recibe los elementos del stream y estima su frecuencia, utilizando para ello la función de hash MurmurHash3. Los elementos con sus frecuencias son almacenados en una estructura que contiene colas de prioridad, específicamente 1024 colas de prioridad, cada una almacenando hasta 8 elementos, de modo que se almacenan los top-8192 elementos del stream en cada momento (en realidad una aproximación a los top-8192 elementos). Los elementos del stream a su vez pasan por una estructura HyperLogLog de tamaño 2^{13} para estimar la cardinalidad del stream de datos. Una vez que se procesa todo el stream, se procede a estimar la entropía. Para ello, utilizamos los top-8192 elementos de las colas de prioridad para estimar la contribución a la entropía de los elementos más frecuentes mediante la siguiente fórmula:

$$H_{est} = -\sum_{i=1}^{K} \frac{m_i}{M} \log \frac{m_i}{M} \tag{3.6}$$

Donde m_i corresponde a la frecuencia del elemento i, M corresponde a la frecuencia total de elementos en el stream de datos, y K corresponde a la cantidad de elementos más frecuentes, en este caso 8192. Esta entropía calculada corresponde a la contribución a la entropía de los elementos más frecuentes. El histograma de frecuencias del stream de datos tiene forma sesgada a la derecha donde unos pocos elementos se repiten muchas veces, y el resto se repite muy pocas veces. Debemos aplicar una corrección a la entropía estimada de modo que se tome en cuenta la contribución de la parte derecha del histograma, es decir, los elementos que no son parte de los top-8192 elementos. Para ello, aplicamos la siguiente fórmula:

$$H_{est} = H_{est} - \frac{M-L}{M} \log \frac{M-L}{M(N-K)} \tag{3.7}$$

Donde N corresponde a la cardinalidad, y L corresponde al número total de

ocurrencias de los top-8192 elementos, definido mediante la siguiente fórmula:

$$L = \sum_{i=1}^{K} m_i \tag{3.8}$$

Finalmente, normalizamos la entropía mediante la siguiente fórmula:

$$H_{est_{norm}} = \frac{H_{est}}{\log N} \tag{3.9}$$

3.4. Paralelismo en CPU

En la actualidad las CPUs contienen múltiples núcleos. La computación paralela consiste en hacer uso de las hebras de CPU (o GPU) para ejecutar distintos procesos simultáneamente. Una manera de hacer uso de las capacidades de computación paralela es mediante el uso de la interfaz de programación OpenMP, la cual permite añadir concurrencia a programas escritos en C/C++ y Fortran.

Capítulo 4

Desarrollo

4.1. Mejoras de espacio y de tiempo en HyperLogLog

En la implementación de HyperLogLog se introdujeron mejoras en el sketch para optimizar el uso de memoria. Para minimizar los bits utilizados por registro, se utilizó un vector de enteros de 64 bits, donde cada entero almacena 16 registros de 4 bits, de modo que si un entero de 64 bits utiliza los 16 registros disponibles, no se desperdicia espacio. De este modo se consigue ahorrar un 50 % de espacio en comparación a una implementación del sketch que utilice un vector de caracteres o enteros de 8 bits, donde hay un registro de 5 bits por celda, de modo que se desperdician 3 bits por cada registro. O bien, se almacenan 12 registros de 5 bits, de modo que si un entero de 64 bits utiliza los 12 registros disponibles, desperdicia solo 4 bits. Así, se ahorra un 33 % de espacio respecto de una implementación del sketch que utilice un vector de caracteres o enteros de 8 bits. Para la implementación en este informe, utilizamos 5 bits por registro. Esto porque de utilizarse 4 bits, podemos tener a lo más un registro de valor 15, por lo que si se calcula un registro entrante con un valor mayor, debe redondearse a 15, lo cual induce un pequeño error en la estimación de la cardinalidad. El error es relativamente pequeño ya que los valores de $\frac{1}{2^m}$ (que deben ser calculados para determinar la media armónica) con registros mayores a 15 son muy pequeños, y también son poco frecuentes, donde m corresponde al registro. Sin embargo, en el estado del arte la estimación de la entropía no debería tener un error relativo mayor a 3 %[2] y, ya que utilizaremos la estimación de la cardinalidad para calcular

Índice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Valor	87	64	60	58	61	54	48	30	35	29	25	18	17	15	9	4	5	0	1	0	0

Figura 4.1: Representación del vector de frecuencias

la estimación de la entropía, queremos tener un valor de la cardinalidad lo más preciso posible.

Para la estimación de la cardinalidad, se utilizó un vector de frecuencias, donde cada celda indica la frecuencia de un registro, con el registro siendo representado por el índice de la celda. Esto se hizo para evitar calcular múltiples veces la potencia de 2 asociada al registro, al momento de calcular la media armónica. La figura 4.1 muestra una representaión del vector de frecuencias, donde el registro con valor 0 se repite 87 veces en el sketch, el registro 1 se repite 64 veces en el sketch, etc.

El espacio utilizado (en bytes) por la extensión del algoritmo HyperLogLog se calculó de la siguiente manera:

$$S_{HLL} = \left\lceil \frac{2^p}{12} \right\rceil \cdot 8 + 2^p \cdot 16 \tag{4.1}$$

Donde S_{HLL} corresponde al espacio utilizado, y se calcula el techo de la división del tamaño del sketch por 12, lo que indica cuantas celdas habrán en el vector de enteros de 64 bits que representan al sketch HyperLogLog. Se multiplica por 8 ya que corresponde a la cantidad de bytes utilizados por un entero de 64 bits. Luego, el resto del sketch está compuesto por el indicador auxiliar, representado por un entero, la frecuencia, representada por un entero, y la demografía, representada por un entero de 64 bits. El tamaño total de estos tres elementos es de 16 bytes.

4.2. Estimación de entropía con múltiples sketches

Se probó guardar una muestra del stream en cada celda del sketch HyperLogLog mediante el uso de sketches CountMin-CU, de modo que cada celda contiene un sketch CountMin-CU y un conjunto de colas de prioridad de tamaño fijo. Para este caso, se utiliza la función de hash MurmurHash3, un sketch HyperLogLog con p = {6,7,8} y los sketches CountMin-CU con d=8 y w={256,128,64}, respectivamente. La idea consiste en guardar una

muestra y quedarse con los top-k, donde estos estarán determinados por colas de prioridad, donde la frecuencia de los elementos guardados en cada cola estará determinada por el sketch CountMin-CU asociado al índice de la cola de prioridad. La frecuencia calculada de los top-k se utilizará posteriormente para calcular la entropía mediante las fórmulas entregadas en las ecuaciones 3.6, 3.7 y 3.9, mientras que la cardinalidad se calculará mediante el sketch HyperLogLog. Es importante notar que cada sketch CountMin-CU tiene un conjunto de 8 colas de prioridad asociadas. En total, los elementos de todas las colas de prioridad asociadas a todos los sketches CountMin-CU suman 8192 elementos, es decir, las colas de prioridad contienen los top-8192 elementos. Esta misma idea se aplica al uso de múltiples sketches TowerSketch-CU, en vez de un único sketch TowerSketch-CU global.

El espacio utilizado por esta implementación está dado por la siguiente fórmula:

$$S_{CMCUs} = \left\lceil \frac{2^p}{12} \right\rceil \cdot 8 + 4 \cdot d \cdot w + 12 \cdot k \tag{4.2}$$

Donde se calcula el techo de la división del tamaño del sketch por 12, lo que indica cuantas celdas habrán en el vector de enteros de 64 bits que representan al sketch HyperLogLog. Se multiplica por 8 ya que corresponde a la cantidad de bytes utilizados por un entero de 64 bits. El tamaño de la matriz del sketch CountMin-CU está dado por d y w, donde d indica la cantidad de hashes a aplicarse a cada elemento. Cada elemento dentro del sketch CountMin-CU es un entero, por lo que cada celda utiliza 4 bytes. El conjunto de colas de prioridad en su totalidad poseen k elementos, donde para cada dirección IP se guarda esta, representada por un entero de 64 bits, y su frecuencia, representada por un entero de 32 bits, por lo que en total se utilizan 12 bytes por dirección IP guardada en las colas de prioridad. El algoritmo de inserción se muestra en el algoritmo 16.

4.3. Estimación de la entropía utilizando HyperLogLog con muestreo

Ya que es posible utilizar la extensión del algoritmo HyperLogLog para obtener una muestra de un conjunto de elementos, se probó utilizar esta funcionalidad para estimar la entropía del conjunto. La idea es utilizar las frecuencias estimadas del sketch para estimar el histograma de frecuencias

Algoritmo 16 Inserción de elementos

```
1: Input stream
2: Output M_i
3: M_i ← 0
4: for ip ∈ stream do
5:     M_i ← M_i + 1
6:     hash ← MurmurHash3 de 64 bits de ip, recortado a 32 bits
7:     v1 ← hash&((1 ≪ p) − 1)
8:     mutex[v1].lock()
9:     hll.insert(hash)
10:    cmcu[v1].insert(ip)
11:    est ← cmcu[v1].estimate(ip)
12:    pq[v1].add(ip,est)
13:    mutex[v1].unlock()
14: end for
```

de la muestra, y luego utilizar dicho histograma para estimar la entropía del conjunto. El algoritmo recibe el arreglo de frecuencias F, el arreglo de demografías D, la cardinalidad N (estimada por el mismo sketch HyperLogLog) y el tamaño M del stream. El algoritmo 17 muestra la estimación de la entropía mediante este método.

También se probó utilizar el algoritmo en conjunto con el sketch CountMin-CU y sus colas de prioridad asociadas, para estimar la contribución a la entropía de los elementos descartados por las colas de prioridad asociadas al sketch CountMin-CU. Para ello, insertamos los elementos que son descartados de las colas de prioridad en la extensión del sketch HyperLogLog, para tomar una muestra de los elementos descartados (es decir, los que no son parte de los top-8192 elementos) y posteriormente estimar su contribución a la entropía. Al momento de estimar la contribución de la entropía por parte de la extensión del sketch HyperLoglog, se utilizan solo los elementos que no están también dentro de las colas de prioridad. La figura 4.2 muestra una representación del algoritmo, donde un elemento del stream se inserta en el sketch HyperLogLog, para estimar la cardinalidad, y en el sketch CountMin-CU, el cual a su vez inserta el elemento y su frecuencia estimada en la cola de prioridad respectiva. Si la cola de prioridad esta llena, descarta el elemento menos frecuente y lo inserta en el sketch HyperLoglog con muestreo. Se utilizan tanto la cola de prioridad como el sketch HyperLogLog con muestreo

Algoritmo 17 Estimación de entropía

Input F, D, N, M
Output H
$T \leftarrow$tabla hash vacía
$tam \leftarrow M.size()$
for $j \leftarrow 0$ to N **do**
    if $F[j] = 0$ **then**
        continue
    end if
    if $D[j]$ in T **then**
        $T[D[j]] \leftarrow T[D[j]] + F[j]$
    else
        $T[D[j]] \leftarrow F[j]$
    end if
end for
$H \leftarrow 0$
for t in T **do**
    $H \leftarrow H + \frac{t}{M} \cdot \log \frac{t}{M}$
end for
$H \leftarrow \frac{H}{\log N}$

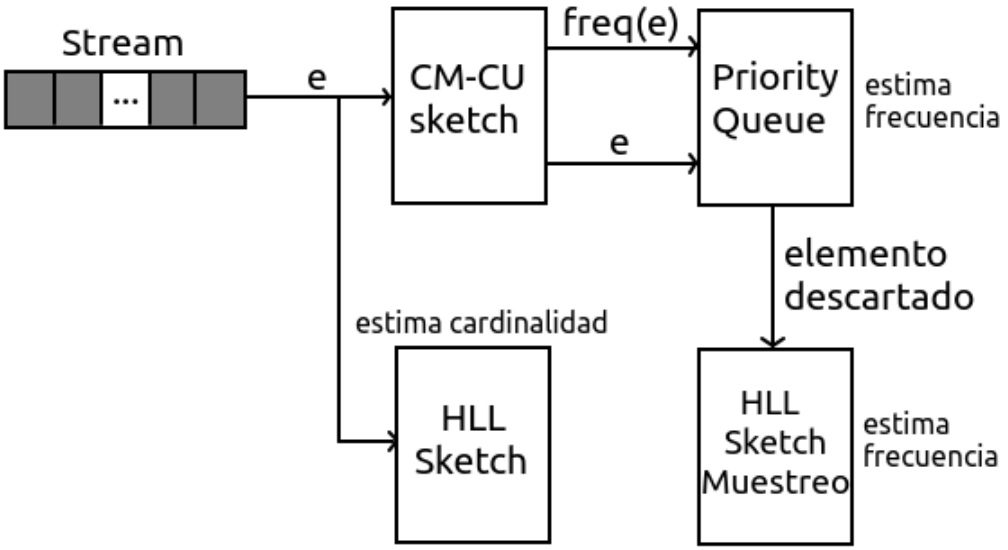


Figura 4.2: Representación del uso del algoritmo HyperLogLog con muestreo en conjunto con el sketch CountMin-CU para estimar la entropía

para estimar las frecuencias de los elementos. El algoritmo 18 muestra la inserción de los elementos mediante este método.

4.4. Estimación de entropía usando paralelismo

Se determinó que solo valía la pena paralelizar la lectura del archivo. No se paralelizaron las implementaciones de los algoritmos CountMin-CU o HyperLogLog, ya que la implementación del algoritmo CountMin-CU ya fue previamente paralelizada[7], sin mejoras en el tiempo de ejecución, y la implementación del algoritmo HyperLogLog solo puede ser paralelizada en la estimación de la cardinalidad, cuyo cálculo de por si ya es muy rápido, por lo que se concluyó que no se obtendrían mejoras al paralelizarlo. Primero se probó leer paralelamente con fscanf mediante punteros que apuntaran a divisiones del archivo. El problema de esto es que la lectura secuencial de un archivo está optimizada a nivel de caché, por lo que resultados preliminares indicaron que este método era de hecho más lento que la lectura secuencial.

Algoritmo 18 Inserción de elementos

Input stream,p, b
Output $M1, M2, M, I, F, D$
$M1 \leftarrow 0$
$M2 \leftarrow 0$
$N \leftarrow 2^p$
Inicializar N registros $M[0], ..., M[N-1]$ en 0
Inicializar N indicadores auxiliares $I[0], ..., I[N-1]$ en 0
Inicializar N contadores de frecuencia $F[0], ..., F[N-1]$ en 0
Inicializar N muestras demográficas $D[0], ..., D[N-1]$ en *null*
for $ip \in$ stream **do**
    $M1 \leftarrow M1 + 1$
    hash $\leftarrow$ Murmurhash de 64 bits de ip, recortado a 32 bits
    hll.insert(hash)
    cmcu.insert(ip)
    est $\leftarrow$ cmcu.estimate(ip)
    rest $\leftarrow$ pq.add(ip,est)
    if rest **then**
        $M2 \leftarrow M2 + 1$
        $x \leftarrow h(e)$
        $j \leftarrow \langle x_0, ..., x_{p-1} \rangle_2$
        $i \leftarrow \langle x_p, ..., x_{p+b-1} \rangle_2$
        $w \leftarrow (\min\{t | t \geq 0 \text{ and } t < b, x_{p+t} = 1\}) + 1$
        if $M[j] < w$ or $(M[j] = w$ and $I[j] < i)$ **then**
            $I[j] \leftarrow i$
            $F[j] \leftarrow 1$
            $D[j] \leftarrow e$
        else if $M[j] = w$ and $I[j] = i$ **then**
            $F[j] \leftarrow F[j] + 1$
            if $D[j] \neq e$ **then**
                Se escoge $D[j]$ de $(D[j], e)$ a través de un criterio predefinido
            end if
        end if
        $M[j] \leftarrow \max(M[j], w)$
    end if
end for

Se probó también insertar paralelamente mediante 2 hebras, una hebra en el sketch HyperLogLog y la otra hebra en el sketch CountMint-CU y en la cola de prioridad correspondiente. El problema de este método es que la inserción en el sketch HyperLogLog es muy rápida, por lo que resultados preliminares no mostraron diferencia alguna en el tiempo. Se probó también pasar el archivo a una versión binaria y luego copiar cada elemento de dicho archivo a memoria de manera iterativa. Esta lectura se puede realizar de manera paralela, conociéndose de antemano la cantidad de lineas a leer por hebra para así hacer las inserciones al sketch HyperLogLog, sketch CountMin-CU y Priority Queue, por lo que se deben utilizar locks para evitar condiciones de carrera. El problema es que configurar los locks toma tiempo. Se utilizó una implementación de mutex más rápida que la ofrecida por la biblioteca std. Además, como se indicó, se requiere conocer el número de lineas a leer. Esto puede hacerse de manera manual o automáticamente, tomando el tamaño total del archivo binario y dividiéndolo por el tamaño de un entero de 64 bits, ya que el archivo binario consiste solamente de dichos enteros. El archivo debe ser pasado a una versión binaria ya que no hay forma eficiente de dividir el archivo en partes sin accidentalmente dividir alguna de las trazas dentro del archivo, ya que cada cifra constituye un caracter dentro del archivo, y las trazas tienen una cantidad distinta de cifras, mientras que en la versión binaria, para un tipo de dato o entero, todos los enteros del mismo tipo tendrán la misma cantidad de caracteres. El algoritmo 19 presenta el pseudocódigo de esta solución.

Donde el algoritmo recibe el tamaño del trozo del archivo a leer, el valor de p para el tamaño del sketch HyperLogLog, la id de la hebra, y la cantidad de elementos a guardarse en memoria a la vez.

Algoritmo 19 Lectura paralela

1: **Input** $chunk$, p, id, etr
2: **Output** M_i
3: $M_i \leftarrow 0$
4: $cicles \leftarrow \frac{chunk}{etr}$
5: **for** $i \leftarrow 0$ to $chunk$ **do**
6: ips ← Trozo del archivo en la posicion i con offset $chunk \cdot id + i \cdot etr$
7: **for** $j \leftarrow 0$ to $cicles$ **do**
8: $M_i \leftarrow M_i + 1$
9: $ip \leftarrow ips[j]$
10: hash ← Murmurhash de 64 bits de ip, recortado a 32 bits
11: v1 ← hash&$((1 \ll p) - 1)$
12: mutex[v1].lock()
13: hll.insert(hash)
14: cmcu[v1].insert(ip)
15: est ← cmcu[v1].estimate(ip)
16: pq[v1].add(ip,est)
17: mutex[v1].unlock()
18: **end for**
19: **end for**

Capítulo 5

Experimentos y Resultados

Las pruebas se realizaron en un entorno Linux de 64 bits con un procesador Intel i5-7200U de 2 núcleos y 4 hebras a una frecuencia máxima de 3.1 GHz y 12GB de RAM.

Se midieron el tiempo de ejecución, el uso de memoria, el error relativo en el cálculo de la entropía, y la aceleración y eficiencia en la ejecución paralela de un algoritmo respecto a su contraparte secuencial. Se calculó también el histograma de frecuencias estimadas y reales de diferentes streams de datos

El uso de memoria se definió mediante el maximum resident set (MRS), el cual corresponde al valor máximo de la cantidad total de memoria física asignada a un proceso durante su ejecución. En el sistema operativo Linux se puede calcular el valor del MRS de un programa llamando al comando time, el cual devolverá el MRS en KBytes.

El error relativo corresponde a la diferencia del error absoluto de la estimación respecto al valor real, dividido por el valor real. Su valor es un porcentaje. El error relativo se muestra en las tablas como E_r. Se calcula mediante la siguiente fórmula:

$$E_r = \frac{|V_{real} - V_{est}|}{V_{real}} \tag{5.1}$$

La aceleración (S) corresponde a qué tan rápido es un algoritmo paralelo respecto de su contraparte secuencial. Se calcula experimentalmente como la razón entre el tiempo de ejecución del algoritmo secuencial con el tiempo de ejecución del algoritmo paralelo. Se calcula mediante la siguiente fórmula:

$$S = \frac{T(s)}{T(p)} \tag{5.2}$$

La eficiencia (E) corresponde a la aceleración que se obtiene para la cantidad de procesadores utilizados. Se calcula como la razón entre la aceleración y la cantidad de procesadores utilizados. Se calcula mediante la siguiente fórmula:

$$E = \frac{S}{p} \tag{5.3}$$

Por último, el histograma de frecuencias corresponde a un gráfico discreto que representa la distribución de frecuencias de datos numéricos. Los histogramas de frecuencias sirven para visualizar las frecuencias de los datos de un conjunto, y poder apreciar aproximadamente donde se produce el peak de las frecuencias.

El cuadro 5.1 muestra las características de los archivos, con la clave utilizada en el gráfico de entropías calculadas. El valor M indica la cantidad de direcciones IP en total, mientras que el valor N indica la cantidad de direcciones IP únicas. Se muestra además la entropía real de las trazas de red.

Cuadro 5.1: Información de los archivos

Traza	Clave	M	N	Entropía real
Chicago-20080319	a	3938620	167768	0.8044
Chicago-20080515	b	12242153	199412	0.7567
Chicago-20110608	c	27046415	393237	0.6942
Chicago-20150219	d	15962529	252239	0.5932
Chicago-20160121	e	31197996	135269	0.6474
Mawi-20161201	f	94738985	5083756	0.4480
Mawi-20171101	g	115486662	4145741	0.4374
Mawi-20181201	h	76066889	4674492	0.4041
Mawi-20191102	i	62478146	4633485	0.4061
Mawi-20200901	j	119923871	5394529	0.3660
Mendeley	k	2668027	101833	0.2709
Sanjose-20081016	l	20919377	334579	0.7009

5.1. Tiempo de ejecución y uso de memoria de HyperLogLogLog

Se probó la estimación de la cardinalidad del conjunto de trazas de red para las implementaciones de HyperLogLog y HyperLogLogLog. El propósito es determinar cómo se comporta el algoritmo HyperLogLogLog en cuanto al tiempo de ejecución y uso de memoria (medido como el MRS) de todas las trazas en conjunto. Ambas implementaciones utilizan registros de 5 bits y un vector de enteros de 64 bits. Cada hebra leyó un archivo a la vez. Los resultados de ambas implementaciones se muestran en los cuadros 5.2 y 5.3.

Cuadro 5.2: Resultados tiempo de ejecución y uso de memoria de algoritmo HyperLogLog

p	Tiempo total (s)	Tiempo construcción (s)	MRS (KB)
6	6.88	6.72	3836
7	6.79	6.78	3888
8	6.85	6.83	4124
9	6.93	6.74	4192
10	6.84	6.75	4064
11	6.83	6.54	3932
12	6.83	6.60	4168
13	6.88	6.76	4080
14	6.82	6.62	4296
15	6.92	6.69	4512

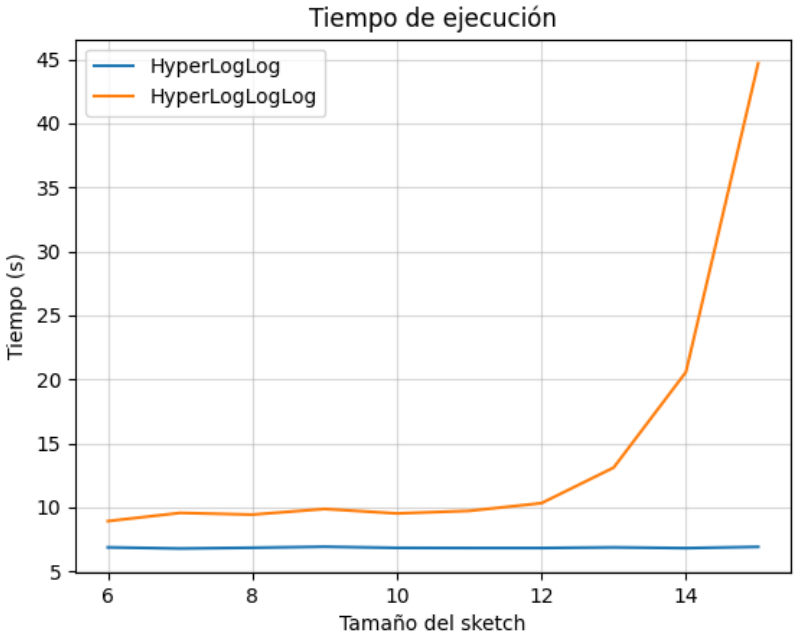


Figura 5.1: Tiempo de ejecución total

Cuadro 5.3: Resultados tiempo de ejecución y uso de memoria de algoritmo HyperLogLogLog

p	Tiempo total (s)	Tiempo construcción (s)	MRS (KB)
6	8.93	8.62	4124
7	9.57	9.31	4252
8	9.44	9.31	4172
9	9.88	9.46	4200
10	9.53	9.48	4116
11	9.73	9.50	4224
12	10.33	10.16	4292
13	13.11	12.27	4440
14	20.57	18.07	4620
15	44.68	43.53	5084

Gráficamente, podemos representar el tiempo total de los algoritmos como se muestra en la figura 5.1, donde vemos que el algoritmo HyperLogLogLog

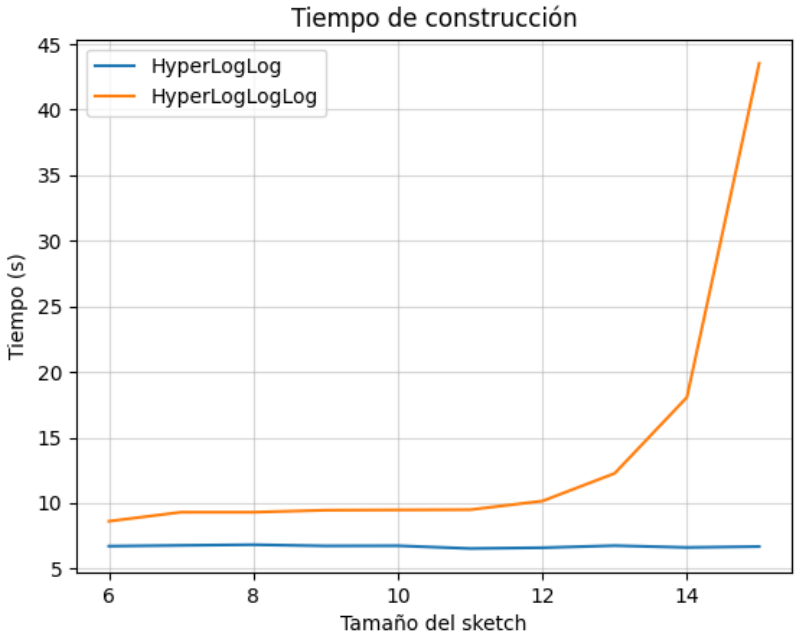


Figura 5.2: Tiempo de construcción

toma significativamente más tiempo que el algoritmo HyperLogLog, y que además tiene un crecimiento más bien cuadrático, mientras que el algoritmo HyperLogLog toma tiempo constante para los tamaños de sketches utilizados, posiblemente debido a que los sketches son demasiado pequeños para apreciarse un uso significativo de tiempo de CPU al momento de estimar la cardinalidad. El tiempo cuadrático del algoritmo HyperLogLogLog se debe a que en el peor caso, cada vez que se inserta un elemento se debe cambiar el valor de base, teniendo que reordenarse el sketch comprimido y la tabla hash.

Podemos representar gráficamente el tiempo de construcción de los algoritmos como se muestra en la figura 5.2, donde vemos que nuevamente que el algoritmo HyperLogLogLog toma significativamente más tiempo que el algoritmo HyperLogLog, y el tiempo de HyperLogLog es mas bien constante para los tamaños de sketch utilizados, mientras que para HyperLogLogLog es exponencial. Además, el tiempo de construcción del sketch es lo que más tiempo toma en ambas implementaciones.

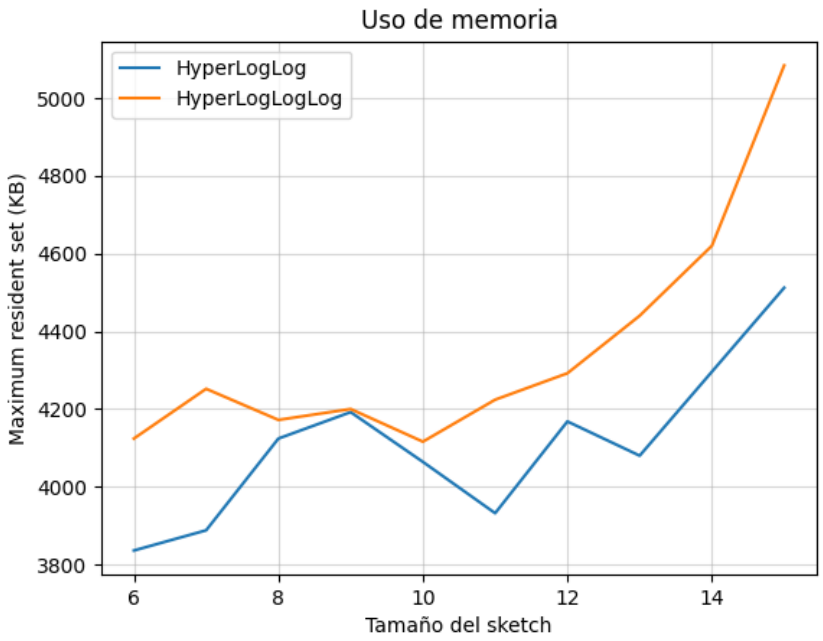


Figura 5.3: Uso de memoria

Podemos representar gráficamente el uso de memoria de los algoritmos como se muestra en la figura 5.3, donde vemos que tanto el algoritmo HyperLogLogLog como el algoritmo HyperLogLog tienen una curva más bien creciente para los tamaños de sketches más grandes, pero que el sketch HyperLogLogLog utiliza ligeramente más memoria que el sketch HyperLogLog.

De esto concluimos que el algoritmo HyperLogLogLog no implementa ninguna mejora a nuestra implementación de HyperLogLog de 5 bits y con vector de enteros de 64 bits. El uso de memoria superior de la implementación HyperLogLogLog se debe al uso de la tabla hash S, la cual almacena los elementos que no están en la región densa y que por lo tanto no van en el sketch comprimido M. Esto porque la tabla hash almacena clave y valor, donde la clave debe ser un entero de 4 bytes y el valor es un caracter de 1 byte, lo cual desperdicia espacio significativo comparado con la implementación HyperLogLog, donde el sketch utiliza menos de 1 byte por registro. Además, el sketch comprimido es de tamaño estático, por lo que siempre se tiene un vector de largo 2^p, donde cada celda utiliza al menos κ bits.

Cuadro 5.4: Error relativo de la estimación de la entropía para múltiples sketches CountMin-CU con w={256,128,64}

Traza	E_r p=6	E_r p=7	E_r p=8
Chicago-20080319	0.0160	0.0278	0.0261
Chicago-20080515	0.0233	0.0292	0.0319
Chicago-20110608	0.0241	0.0290	0.0264
Chicago-20150219	0.0211	0.0082	0.0130
Chicago-20160121	0.0206	0.0092	0.0060
Mawi-20161201	0.0059	0.0031	0.0097
Mawi-20171101	0.0127	0.0084	0.0087
Mawi-20181201	0.0017	0.0078	0.0042
Mawi-20200901	0.0172	0.0136	0.0095
Mawi-20191102	0.0108	0.0099	0.0123
Mendeley	0.0116	0.0047	0.0023
Sanjose-20081016	0.0420	0.0310	0.0314

5.2. Estimación de entropía, uso de memoria y tiempo de ejecución con múltiples sketches CountMin-CU

Para la experimentación con HyperLogLog con CountMin-CU utilizamos las siguientes configuraciones:

- p=6, d=8, w=256
- p=7, d=8, w=128
- p=8, d=8, w=64

Donde el valor total de w en cada implementación es de 16384, y se calcula como el valor de w multiplicado por el tamaño del sketch HyperLogLog.

En el cuadro 5.4 se muestran los resultados de los errores relativos de las entropías calculadas para las configuraciones anteriores

Revisaremos también los resultados de las entropías estimadas con un solo CountMin-CU (con 8 colas de prioridad asociadas, cada una almacenando 1024 elementos), para las siguientes configuraciones:

- p=6, d=8, w=16384
- p=7, d=8, w=16384
- p=8, d=8, w=16384

En este caso, el valor total de w es el indicado en las configuraciones, ya que hay solo un sketch CountMin-CU, en vez de uno por celda del sketch HyperLogLog.

En el cuadro 5.5 se muestran los errores relativos obtenidos para las configuraciones indicadas.

Cuadro 5.5: Error relativo de la estimación de la entropía para sketch global CountMin-CU con w=16384

Traza	E_r p=6	E_r p=7	E_r p=8
Chicago-20080319	0.0158	0.0285	0.0269
Chicago-20080515	0.0233	0.0292	0.0315
Chicago-20110608	0.0241	0.0289	0.0265
Chicago-20150219	0.0212	0.0083	0.0131
Chicago-20160121	0.0206	0.0091	0.0061
Mawi-20161201	0.0066	0.0034	0.0099
Mawi-20171101	0.0127	0.0084	0.0087
Mawi-20181201	0.0017	0.0078	0.0042
Mawi-20200901	0.0171	0.0136	0.0095
Mawi-20191102	0.0108	0.0099	0.0123
Mendeley	0.0117	0.0047	0.0023
Sanjose-20081016	0.0421	0.0311	0.0312

Gráficamente podemos representar ambos resultados como se muestra en la figura 5.4. Comparando los resultados de las implementaciones con múltiples sketches CountMin-CU y con un único sketch CountMin-CU, vemos que ambas implementaciones tienen aproximadamente los mismos errores relativos, pero con pequeñas diferencias. Esto posiblemente debido a que se utilizan más colas de prioridad, ya que al tener cada cola de prioridad un límite en la cantidad de elementos a almacenar, se deben ir descartando los elementos con menor frecuencia al llegar un elemento con frecuencia mayor cuando estas se llenan, de modo que las colas de prioridad almacenan una

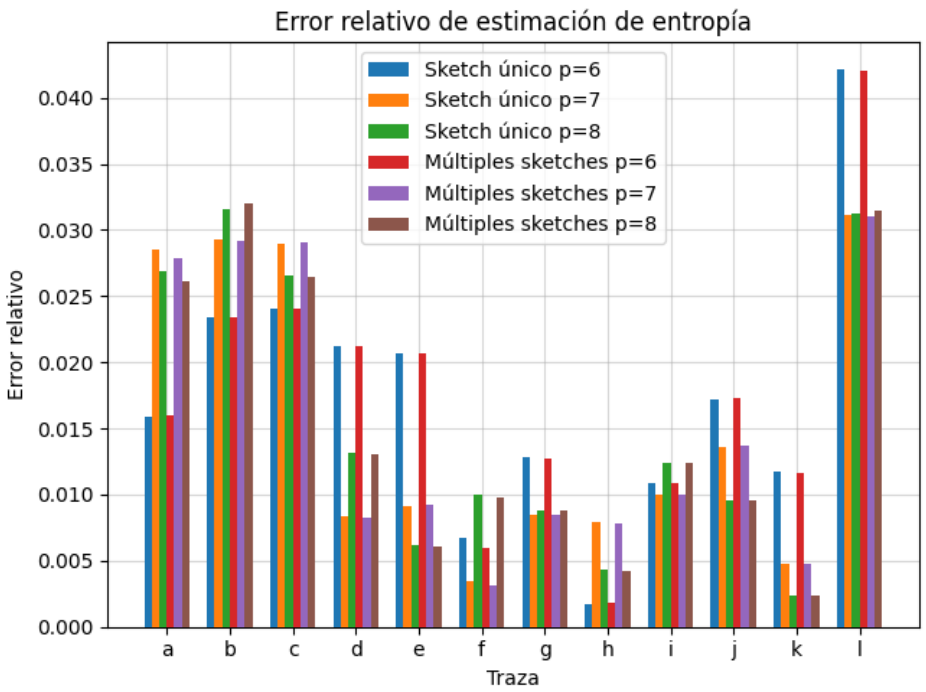


Figura 5.4: Error relativo de la estimación de la entropía con un único sketch CountMin-CU, y múltiples sketches CountMin-CU

Cuadro 5.6: Error relativo estimación de contribución a la entropía de los top-8192 elementos con múltiples sketches CountMin-CU con w={256,128,64}

Traza	E_r p=6	E_r p=7	E_r p=8
Chicago-20080319	0.0305	0.0110	0.0132
Chicago-20080515	0.0270	0.0186	0.0175
Chicago-20110608	0.0204	0.0144	0.0170
Chicago-20150219	0.0027	0.0191	0.0130
Chicago-20160121	0.0027	0.0098	0.0127
Mawi-20161201	0.0048	0.0081	0.0005
Mawi-20171101	0.0069	0.0020	0.0022
Mawi-20181201	0.0027	0.0093	0.0049
Mawi-20191102	0.0087	0.0075	0.0109
Mawi-20200901	0.0089	0.0047	0.0004
Mendeley	0.0249	0.0140	0.0104
Sanjose-20081016	0.0121	0.0034	0.0044

estimación de los top-k. Al utilizarse más colas de prioridad, y al estar estas asociadas al índice de la celda del sketch HyperLogLog a la que una dirección IP llegará, se produce una distinta estimación de los top-k, ya que puede ocurrir que una dirección IP de alta frecuencia sea eliminada de una de las 8 colas de prioridad en la implementación con sketch CountMin-CU único y reemplazada por otra dirección IP de menor frecuencia y cuya celda con el registro en el sketch HyperLogLog utilice un índice diferente a la dirección IP de mayor frecuencia. Esto no ocurriría en la implementación con mútliples sketches CountMin-CU, ya que esas direcciones IP irían a diferentes colas de prioridad basadas en el índice de la celda del sketch HyperLogLog.

Revisaremos ahora los resultados de la contribución a la entropía de los top-8192 elementos para la implementación con múltiples CountMin-CU, comparado con la implementación con CountMin-CU único. Para ello utilizaremos las configuraciones p={6,7,8}, d=8 y w={256,128,64} (en el caso de CountMin-CU único se tiene w=16384). Los resultados se muestran en los cuadros 5.6 y 5.7.

Gráficamente podemos representar ambos resultados como se muestra en la figura 5.5, donde vemos nuevamente que los resultados de ambos son similares, con pequeñas diferencias debido al uso de más colas de prioridad en la implementación con múltiples CountMin-CU. El error relativo en ambos

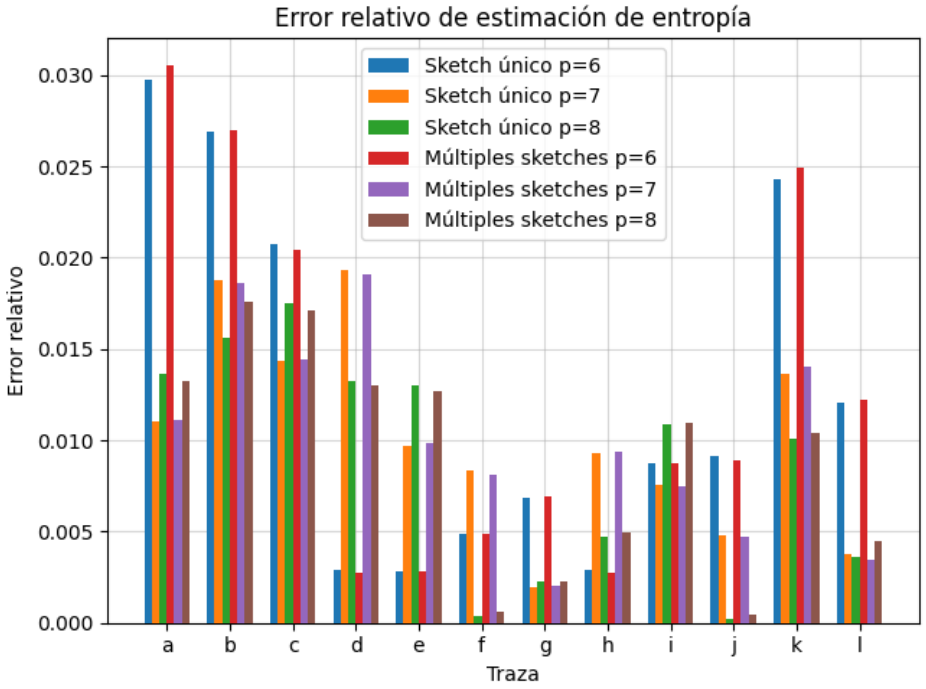


Figura 5.5: Error relativo de la estimación de la contribución a la entropía de los top-8192 elementos con un único sketch CountMin-CU, y múltiples sketches CountMin-CU

Cuadro 5.7: Error relativo estimación de contribución a la entropía de los top-8192 elementos con único sketch CountMin-CU con w=16384

Traza	E_r p=6	E_r p=7	E_r p=8
Chicago-20080319	0.0297	0.0110	0.0136
Chicago-20080515	0.0268	0.0187	0.0156
Chicago-20110608	0.0207	0.0143	0.0175
Chicago-20150219	0.0029	0.0193	0.0132
Chicago-20160121	0.0028	0.0096	0.0130
Mawi-20161201	0.0048	0.0083	0.0003
Mawi-20171101	0.0068	0.0019	0.0022
Mawi-20181201	0.0028	0.0092	0.0047
Mawi-20191102	0.0087	0.0075	0.0109
Mawi-20200901	0.0091	0.0047	0.0002
Mendeley	0.0242	0.0136	0.0100
Sanjose-20081016	0.0120	0.0037	0.0036

casos es relativamente pequeño, por lo que la estimación de la entropía de los top-8192 elementos es bastante precisa.

Se probó además utilizar un CountMin-CU por hebra, con sus colas de prioridad asociadas, en vez de utilizar un CountMin-CU por elemento del sketch HyperLogLog. Una vez procesado el stream se hace merge de las colas de prioridad. Se determinó que utilar este enfoque no es eficiente, ya que cada CountMin-CU debe tener asociado 1024 colas de prioridad de 8 elementos cada una (es decir, cada CountMin-CU tiene asociado los top-8192 elementos). Esto debido a que no se sabe en que hebra caerán las direcciones IP, y puede ocurrir que, al momento de hacerse merge, debido a que múltiples colas tenían contadores de la misma dirección IP, queden menos de 8192 elementos, lo cual fue comprobado mediante la experimentación.

A continuación revisaremos el uso de memoria de la implementación con múltiples CountMin-CU, para las configuraciones anteriores. Se midió el uso de memoria mediante el comando time, el cual devolvió el MRS, medido en KBytes. Los resultados se muestran en el cuadro 5.8.

Cuadro 5.8: MRS (KB) con múltiples sketches CountMin-CU con w={256,128,64}

Traza	MRS p=6	MRS p=7	MRS p=8
Chicago-20080319	5244	5688	5804
Chicago-20080515	5268	5752	5808
Chicago-20110608	5292	5832	5904
Chicago-20150219	5356	5848	5908
Chicago-20160121	5416	5844	5952
Mawi-20161201	5496	5716	5804
Mawi-20171101	5552	5852	6004
Mawi-20181201	5420	5800	5944
Mawi-20191102	5364	5792	5900
Mawi-20200901	5300	5688	5812
Mendeley	5316	5644	5764
Sanjose-20081016	5388	6016	5936

El cuadro muestra que el tamaño del MRS aumenta junto con el valor de p, a pesar de que la cantidad total de celdas de los sketches CountMin-CU permanece constante.

A continuación revisaremos el uso de memoria de la implementación con CountMin-CU único, para las configuraciones anteriores. Se midió el uso de memoria mediante el comando time, el cual devolvió el MRS, medido en KBytes. Los resultados se muestran en el cuadro 5.9.

Cuadro 5.9: MRS (KB) con sketch único CountMin-CU con w=16384

Traza	MRS p=6	MRS p=7	MRS p=8
Chicago-20080319	4892	4812	4964
Chicago-20080515	4952	4828	4932
Chicago-20110608	4952	4904	4900
Chicago-20150219	4984	4756	4892
Chicago-20160121	4916	4728	4900
Mawi-20161201	4892	4868	4952
Mawi-20171101	4948	4816	5000
Mawi-20181201	5004	4816	5016
Mawi-20191102	4892	4788	5004
Mawi-20200901	4952	4868	5008
Mendeley	4952	4872	4896
Sanjose-20081016	4888	4776	4896

El cuadro muestra que no hay diferencia significativa en el uso de memoria entre las distintas configuraciones para la implementación con CountMin-CU único, pero si hay diferencias en el uso de memoria entre la implementación con CountMin-CU único y la implementación con múltiples CountMin-CU. Debido a que en esta implementación no hay diferencia significativa en el uso de memoria para distintos tamaños del sketch HyperLogLog, y la implementación del sketch HyperLogLog es la misma para ambas implementaciones, luego el tamaño del sketch HyperLoglog no es la razón por la cual hay diferencias en el uso de memoria para las distintas configuraciones en la implementación con múltiples sketches CountMin-CU.

A continuación mostraremos el uso de memoria de la implementación con múltiples sketches CountMin-CU y sin uso de locks (por lo que no puede ser ejecutado concurrentemente). Recordemos que se utiliza un lock por celda del sketch HyperLogLog. Los resultados se muestran en el cuadro 5.10.

Cuadro 5.10: MRS (KB) con múltiples sketches CountMin-CU w={256,128,64} y sin locks

Traza	MRS p=6	MRS p=7	MRS p=8
Chicago-20080319	5192	5052	5072
Chicago-20080515	4996	5092	5176
Chicago-20110608	5056	5140	5076
Chicago-20150219	5100	5188	5124
Chicago-20160121	5052	5096	5196
Mawi-20161201	5136	5132	5208
Mawi-20171101	5052	4988	5224
Mawi-20181201	5192	4988	5072
Mawi-20191102	5144	5128	5136
Mawi-20200901	5116	5184	5416
Mendeley	5188	5332	5132
Sanjose-20081016	5000	5096	5144

El cuadro muestra que el uso de memoria se reduce y además permanece constante para la distintas configuraciones, esto porque un tamaño de sketch HyperLogLog más grande utiliza más locks. Sin embargo, el uso de memoria sigue siendo superior a la implementación con CountMin-CU único. Es importante notar además que la implementación con CountMin-CU único no utiliza locks, ya que al utilizarse un sketch CountMin-CU único, la ejecución no puede utilizar concurrencia sin producir condiciones de carrera. El uso de memoria superior en la implementación con múltiples CountMin-CU se debe al uso de vectores con sketches CountMin-CU y colas de prioridad, debido a la forma en que los vectores reservan memoria.

Por último, calculamos el tiempo de ejecución utilizando múltiples sketches CountMin-CU. Para ello, se llamo al comando time. El tiempo de ejecución se calculó en segundos.

A continuación se presentan los resultados de la ejecución secuencial de la implementación con múltiples CountMin-CU para diferentes configuraciones. Los resultados se muestran en el cuadro 5.11.

El cuadro muestra que el tiempo no varía para las distintas configuraciones. Esto porque el tamaño del sketch HyperLogLog en todos los casos es demasiado pequeño para apreciarse alguna diferencia en el tiempo de ejecución.

Cuadro 5.11: Tiempo de ejecución secuencial con múltiples sketches CountMin-CU con w={256,128,64}

Traza	Tiempo p=6	Tiempo p=7	Tiempo p=8
Chicago-20080319	4.71	4.77	4.75
Chicago-20080515	14.16	14.92	14.17
Chicago-20110608	30.95	31.86	30.96
Chicago-20150219	17.98	18.72	17.99
Chicago-20160121	34.99	35.54	34.97
Mawi-20161201	105.03	106.81	104.77
Mawi-20171101	127.31	129.35	127.14
Mawi-20181201	83.75	86.35	83.70
Mawi-20191102	68.93	71.47	68.95
Mawi-20200901	132.52	133.46	132.01
Mendeley	2.96	3.16	2.95
Sanjose-20081016	24.00	24.52	24.05

También calculamos el tiempo de ejecución secuencial de la implementación con un CountMin-CU para distintas configuraciones. Los resultados se muestran en el cuadro 5.12.

Cuadro 5.12: Tiempo de ejecución secuencial con sketch CountMin-CU global con w=16384

Traza	Tiempo p=6	Tiempo p=7	Tiempo p=8
Chicago-20080319	4.68	4.73	4.61
Chicago-20080515	14.05	14.07	14.08
Chicago-20110608	30.67	30.71	30.81
Chicago-20150219	17.83	17.83	18.22
Chicago-20160121	34.69	34.68	34.70
Mawi-20161201	104.13	106.02	104.11
Mawi-20171101	126.34	126.19	126.09
Mawi-20181201	83.16	83.15	83.16
Mawi-20191102	68.42	68.52	68.48
Mawi-20200901	131.20	131.06	133.07
Mendeley	2.94	2.93	2.95
Sanjose-20081016	23.78	23.78	23.83

Cuadro 5.13: Error relativo para TowerSketch-CU con p=8

Traza	E_r $w = 2^{14}$	E_r $w = 2^{15}$
Chicago-20080319	0.0008	0.0265
Chicago-20080515	0.0157	0.0307
Chicago-20110608	0.0481	0.0236
Chicago-20150219	0.0193	0.0127
Chicago-20160121	0.0177	4.01e-05
Mawi-20161201	76.0665	0.0057
Mawi-20171101	64.6768	0.0044
Mawi-20181201	102.9651	0.0055
Mawi-20191102	0.0922	0.0142
Mawi-20200901	74.9281	0.0031
Mendeley	0.0013	0.0055
Sanjose-20081016	0.0317	0.0304

El cuadro muestra que el tiempo de ejecución no varía para las distintas configuraciones, y es además casi equivalente al tiempo de ejecución secuencial para la implementación con múltiples CountMin-CU. Solo hay una pequeña diferencia debido al tiempo requerido para configurar los locks.

5.3. Estimación de entropía, uso de memoria y tiempo de ejecución con TowerSketch-CU

Se evaluó el uso del algoritmo TowerSketch-CU en reemplazo del algoritmo CountMin-CU. Para ello, se utilizó un único sketch TowerSketch-CU, un valor de p=8, un valor de w=$\{2^{14}, 2^{15}\}$ y un valor de d=5, el cual no pudo ser aumentado, ya que otros parámetros dependen de este valor, los cuales no pueden ser modificados sin que el algoritmo no funcione correctamente. Los resultados de los errores relativos en las estimaciones de las entropías se muestran en el cuadro 5.13.

El cuadro muestra buenos resultados para w=2^{15}, pero no para los archivos Mawi en la configuración con w=2^{14}, donde el error relativo aumenta de manera muy significativa, llegando incluso a superar el 10000 %. Del cua-

Cuadro 5.14: Error relativo con múltiples sketches TowerSketch-CU con w={512,256,128}

Archivo	E_r p=6	E_r p=7	E_r p=8
Chicago-20080319	0.0165	0.0282	0.0266
Chicago-20080515	0.0224	0.0287	0.0311
Chicago-20110608	0.0217	0.0265	0.0237
Chicago-20150219	0.0207	0.0081	0.0134
Chicago-20160121	0.0187	0.0079	0.0040
Mawi-20161201	0.0130	0.0373	0.0247
Mawi-20171101	0.0029	0.0055	0.0213
Mawi-20181201	0.0011	0.0064	0.0004
Mawi-20191102	0.0070	0.0104	0.0096
Mawi-20200901	0.0028	0.0063	0.0078
Mendeley	0.0146	0.0078	0.0055
Sanjose-20081016	0.0414	0.0299	0.0307

dro 5.1 podemos ver que los archivos Mawi contienen los streams de datos más grandes de todos los archivos y, dado que el algoritmo TowerSketch-CU tiende a sobrestimar los contadores para streams de datos grandes, podemos concluir que el error en el cálculo de la entropía estimada se debe a la sobrestimación de los contadores.

Evaluamos utilizar múltiples sketches TowerSketch-CU, con un sketch TowerSketch-CU por celda del sketch HyperLogLog, con sus colas de prioridad asociadas. Utilizaremos la configuración w=2^{15}, ya que es la que produjo los mejores resultados. Probaremos con p={6,7,8} y con ello w={512,256,128}. Los resultados de los errores relativos en las estimaciones de las entropías se muestran en el cuadro 5.14.

El cuadro muestra que los resultados son ligeramente peores a la implementación con un único sketch TowerSketch-CU, pero no con una diferencia significativa. Además, los resultados del uso de múltiples sketches TowerSketch-CU con un tamaño de sketch HyperLogLog p=8 son diferentes al uso de un único sketch TowerSketch-CU con un mismo tamaño de sketch HyperLogLog, al contrario de lo que ocurre con la utilización de múltiples sketches CountMin-CU comparado con un único sketch CountMin-CU. No hay una diferencia significativa en el error relativo respecto de los resultados obtenidos con el uso de múltiples sketches CountMin-CU.

Cuadro 5.15: Maximum resident set (KB) con múltiples sketches TowerSketch-CU con w={512,256,128}

Traza	MRS p=6	MRS p=7	MRS p=8
Chicago-20080319	5360	5500	6076
Chicago-20080515	5184	5676	5924
Chicago-20110608	5528	5480	6072
Chicago-20150219	5512	5536	5860
Chicago-20160121	5356	5524	5932
Mawi-20161201	5532	5672	5868
Mawi-20171101	5388	5540	5892
Mawi-20181201	5312	5480	5896
Mawi-20191102	5324	5456	5924
Mawi-20200901	5188	5488	5924
Mendeley	5184	5464	5920
Sanjose-20081016	5324	5476	6064

Se calculó también el uso de memoria, mediante el MRS, en KBytes, de las configuraciones anteriores para la implementación con múltiples sketches TowerSketch-CU. Los resultados se muestran en el cuadro 5.15, donde podemos apreciar que, al igual que en la implementación con sketches CountMin-CU, el uso de memoria aumenta a medida de que se usan más sketches TowerSketch-CU, lo cual se debe a la mayor cantidad de locks utilizados. El uso de memoria para cada valor de p es similar al uso de memoria para la implementación con sketches CountMin-CU para los mismos valores de p.

Se calculó el error relativo de la estimación de la contribución a la entropía de los top-8192 elementos de las trazas de red mediante el sketch TowerSketch-CU (se utiliza un único sketch). Esto para ver si la estimación de estas entropías es mejor a la estimada por el sketch (o sketches) CountMin-CU. De ser así, la diferencia en el valor de las entropías globales se debe al método, es decir, a la fórmula utilizada para estimar la entropía de los elementos restantes, que son los menos frecuentes. El cuadro 5.16 muestra los errores relativos calculados.

Al comparar con los resultados del cuadro 5.7, vemos que no hay diferencias significativas en los errores relativos, por lo que no hay una diferencia significativa entre utilizar el sketch CountMin-CU o el sketch TowerSketch-CU para estimar la entropía de las trazas de red.

Cuadro 5.16: Errores relativos de estimación de contribución a la entropía de top-8192 elementos con sketch TowerSketch-CU con w=2^{15}

Archivo	E_r p=6	E_r p=7	E_r p=8
Chicago-20080319	0.0327	0.0138	0.0164
Chicago-20080515	0.0241	0.0178	0.0136
Chicago-20110608	0.0113	0.0085	0.0119
Chicago-20150219	0.0036	0.0205	0.0137
Chicago-20160121	0.0046	0.0070	0.0111
Mawi-20161201	0.0319	0.0549	0.0221
Mawi-20171101	0.0161	0.0049	0.0100
Mawi-20181201	0.0003	0.0073	0.0015
Mawi-20191102	0.0058	0.0014	0.0116
Mawi-20200901	0.0285	0.0352	0.0103
Mendeley	0.0003	0.0100	0.0134
Sanjose-20081016	0.0134	0.0014	0.0019

Por último, calculamos el tiempo de ejecución para las distintas trazas de red utilizando un único sketch TowerSketch-CU y w=32768. Los resultados se muestran en el cuadro 5.17. Para calcular el tiempo de ejecución se llamó al comando time. Los resultados se muestran en segundos.

Cuadro 5.17: Tiempo de ejecución con sketch TowerSketch-CU global y w=32768

Traza	Tiempo p=6	Tiempo p=7	Tiempo p=8
Chicago-20080319	6.10	5.92	6.02
Chicago-20080515	18.54	18.44	18.42
Chicago-20110608	40.85	40.52	40.59
Chicago-20150219	23.93	23.77	23.80
Chicago-20160121	46.96	46.31	46.32
Mawi-20161201	142.73	141.52	141.68
Mawi-20171101	173.35	172.63	172.65
Mawi-20181201	110.95	110.46	110.84
Mawi-20191102	93.76	93.13	93.19
Mawi-20200901	179.44	176.43	176.45
Mendeley	4.05	4.06	4.04
Sanjose-20081016	31.76	32.15	31.55

De donde vemos que no hay diferencias significativas en el tiempo de ejecución para distintos tamaños del sketch HyperLogLog. Además, el tiempo de ejecución es alrededor de un 50 % mayor que el tiempo de ejecución con el sketch CountMin-CU. Esto último debido a las operaciones extra que se realizan al momento de insertar un elemento en el sketch TowerSketch-CU (son 21 operaciones extra, principalmente a nivel de bits).

5.4. Estimación del error relativo y absoluto

Se calcularon los errores relativo y absoluto promedio de la estimación de frecuencias de los sketches CountMin (CM), CountMin-CU (CM-CU), TowerSketch (TS), y TowerSketch-CU (TS-CU). Esto para ver como se diferencian los sketches en la estimación de las frecuencias de los elementos de un stream de datos. Se utilizaron los parámetros d=5 y w=2^{18}. Los resultados del error relativo promedio se muestran en el cuadro 5.18, mientras que los resultados del error absoluto promedio se muestran en el cuadro 5.19.

Tanto con el error relativo promedio como con el error absoluto promedio, podemos ver que los sketches TowerSketch tienen mejores resultados que los sketches CountMin, y además los sketches CU tienen mejores resultados que los sketches no CU. La razón por la cual los sketches CountMin tienen

Cuadro 5.18: Error relativo promedio para distintos sketches de estimación de cuentas

Traza	CM	CM-CU	TS	TS-CU
Chicago-20080319	9.0303	5.6439	0.3995	0.1538
Chicago-20080515	14.9216	9.5965	0.9379	0.4478
Chicago-20110608	39.1503	23.9805	4.4776	2.1814
Chicago-20150219	16.2758	9.7682	1.3130	0.5427
Chicago-20160121	8.1608	5.3257	0.4244	0.2117
Mawi-20161201	297.5420	154.5390	101.6310	48.9429
Mawi-20171101	185.7110	98.5860	64.4941	22.7027
Mawi-20181201	203.6530	104.4680	66.4188	24.3181
Mawi-20191102	218.9120	114.0770	72.3183	27.8557
Mawi-20200901	237.7270	141.1140	83.9883	33.9201
Mendeley	2.1899	1.0673	0.0709	0.0141
Sanjose-20081016	31.7503	19.0192	3.8278	1.7910

Cuadro 5.19: Error absoluto promedio para distintos sketches de estimación de cuentas

Traza	CM	CM-CU	TS	TS-CU
Chicago-20080319	17.6136	8.8034	2.5309	0.4399
Chicago-20080515	34.8808	18.6030	8.2468	1.9993
Chicago-20110608	83.1325	45.5140	19.6542	6.5137
Chicago-20150219	38.8308	19.9824	7.4198	1.6956
Chicago-20160121	20.3765	10.4473	6.8672	1.9322
Mawi-20161201	329.2670	169.8590	112.8260	53.4744
Mawi-20171101	193.9490	102.3190	67.8996	23.9760
Mawi-20181201	212.6060	108.5270	69.4881	25.1587
Mawi-20191102	227.7280	117.9550	75.7935	28.6626
Mawi-20200901	245.2750	144.9270	86.9358	34.7241
Mendeley	3.9619	1.6207	0.2721	0.0219
Sanjose-20081016	87.3367	45.6695	21.0862	6.6706

un error relativo y absoluto promedio significativamente más altos que los sketches TowerSketch, es que los sketches CountMin nunca subestiman una frecuencia, pero si pueden sobrestimar, y esto se da de manera más signi-

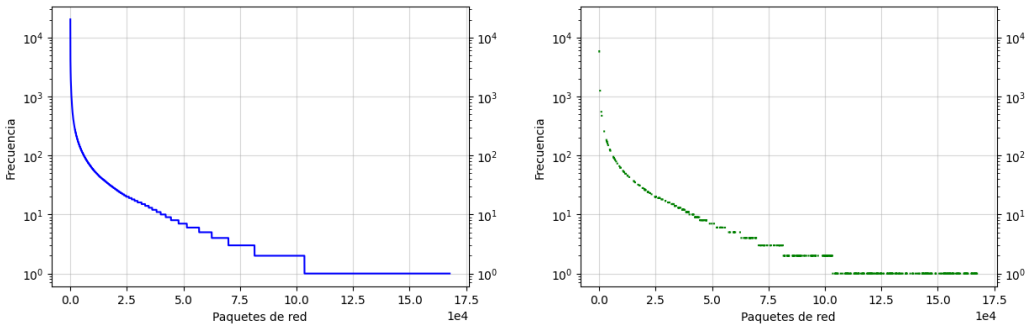
ficativa para las frecuencias de valores pequeños. Ya que las trazas de red poseen pocas direcciones IP que se repiten muchas veces, y muchas direcciones IP que se repiten pocas veces, los valores promedio de los errores relativo y absoluto serán grandes.

5.5. Estimación de frecuencias y entropía con extensión de HyperLogLog

Se calcularon los histogramas de frecuencias reales y estimadas de cada archivo, para ver cómo se comporta el algoritmo en cuanto a la estimación de las frecuencias de las direcciones IP. Luego se procedió a calcular las entropías reales y estimadas de los archivos.

En las figuras 5.6 a 5.17 mostraremos los histogramas de frecuencia de las distintas trazas de red.

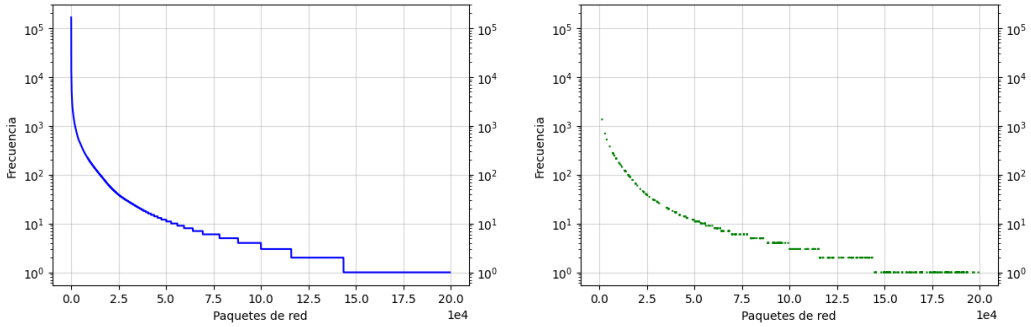
Figura 5.6: Histograma de frecuencias Chicago-20080319



(a) Frecuencias reales

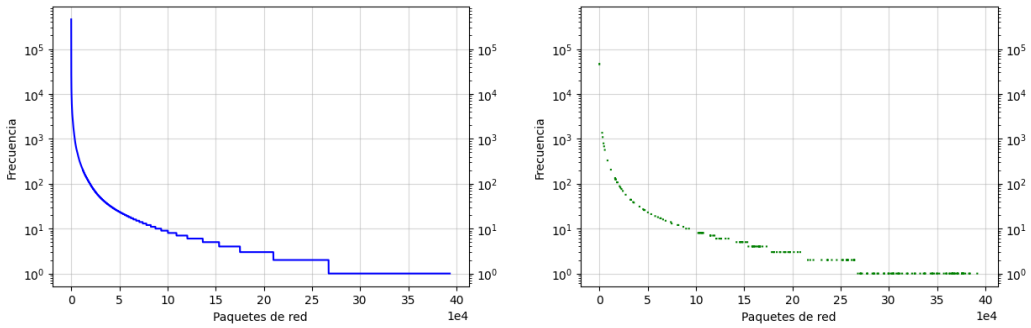
(b) Frecuencias estimadas

Figura 5.7: Histograma de frecuencias Chicago-20080515



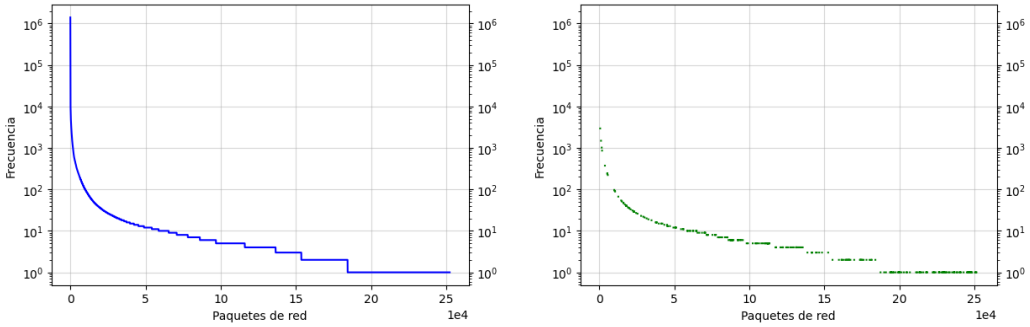
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.8: Histograma de frecuencias Chicago-20110608



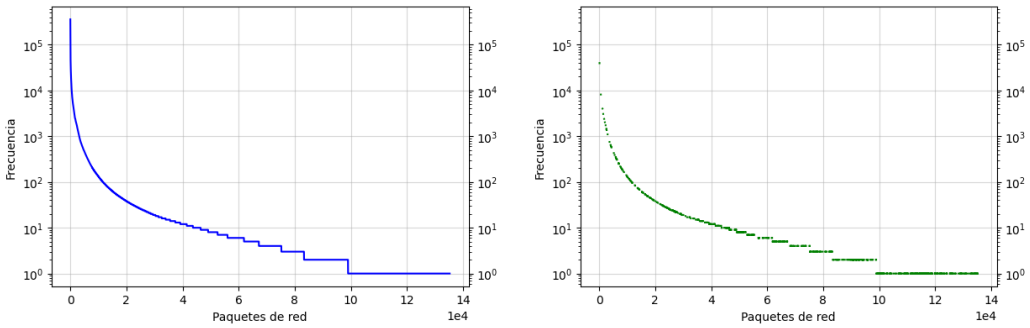
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.9: Histograma de frecuencias Chicago-20150219



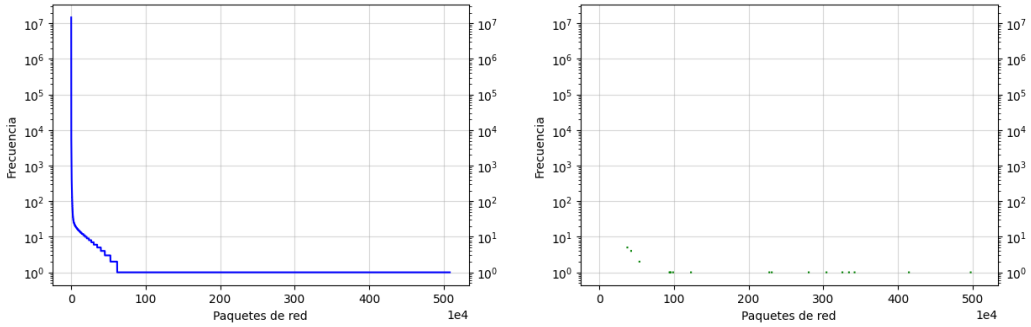
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.10: Histograma de frecuencias Chicago-20160121



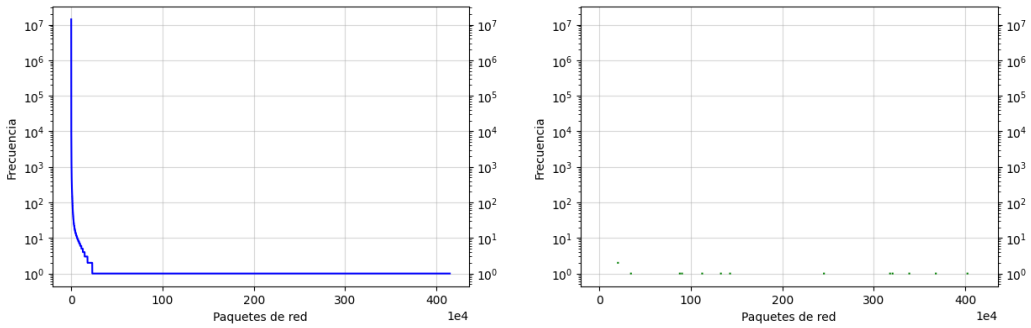
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.11: Histograma de frecuencias Mawi-20161201



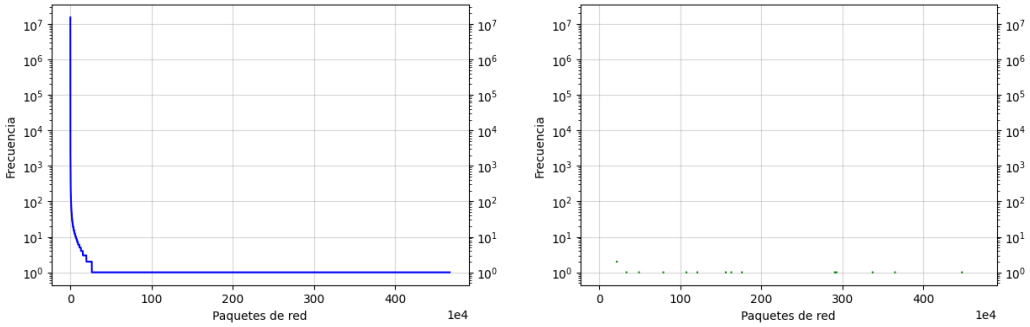
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.12: Histograma de frecuencias Mawi-20171101



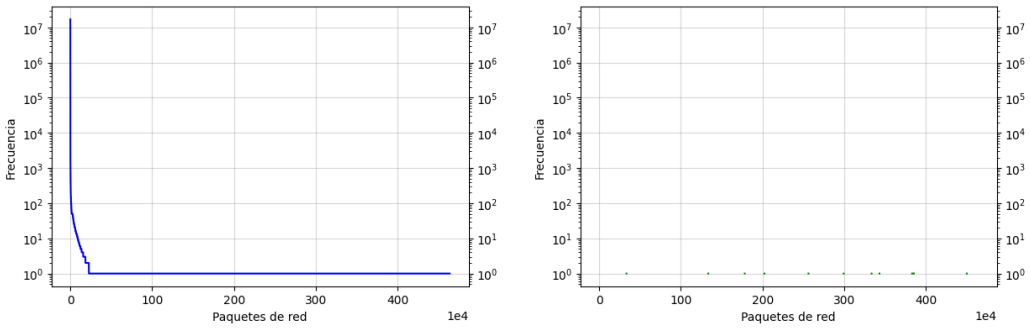
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.13: Histograma de frecuencias Mawi-20181201



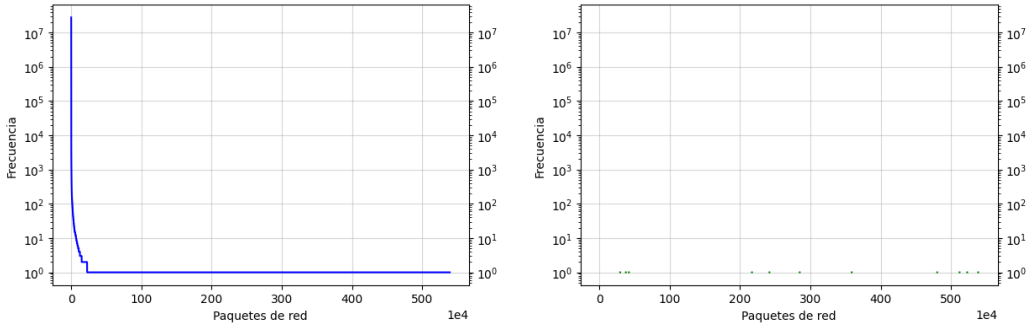
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.14: Histograma de frecuencias Mawi-20191102



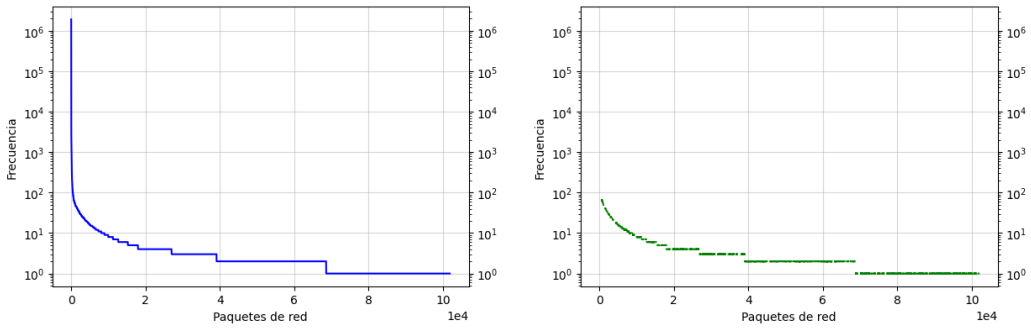
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.15: Histograma de frecuencias Mawi-20200901



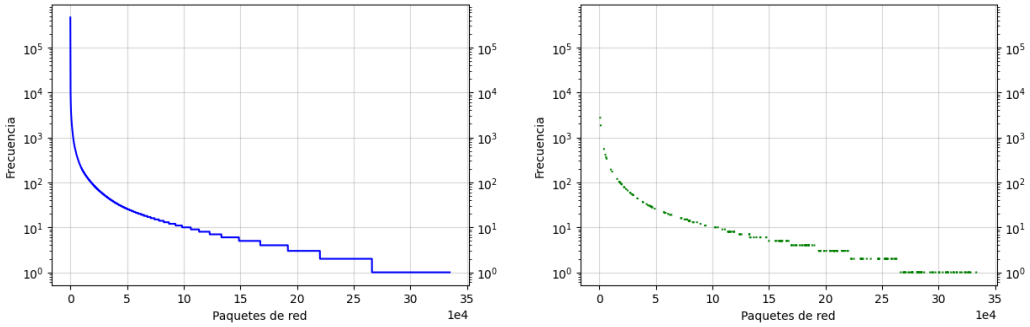
(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.16: Histograma de frecuencias Mendeley



(a) Frecuencias reales (b) Frecuencias estimadas

Figura 5.17: Histograma de frecuencias Sanjose-20081016



(a) Frecuencias reales (b) Frecuencias estimadas

De donde vemos que el algoritmo tiene problemas para captar los elementos más frecuentes de las trazas de red, aunque las gráficas tienen aproximadamente la misma forma, a excepción de los archivos más grandes, ya que la muestra es siempre del mismo tamaño para todos los archivos, pues depende del tamaño del sketch HyperLogLog (hay 1 elemento de la muestra por celda del sketch).

Los resultados de la estimación de la entropía de las trazas para distintos tamaños del sketch se muestran en el cuadro 5.20, la cual utiliza las claves del cuadro 5.1 y p=13 para el tamaño del sketch HyperLogLog.

Cuadro 5.20: Entropía estimada mediante la extensión de HyperLogLog p=13

Traza	Entropía estimada	Error relativo
Chicago-20080319	0.0379	0.9527
Chicago-20080515	0.0295	0.9609
Chicago-20110608	0.0158	0.9771
Chicago-20150219	0.0174	0.9706
Chicago-20160121	0.0365	0.9435
Mawi-20161201	0.0004	0.9990
Mawi-20171101	0.0013	0.9968
Mawi-20181201	0.0006	0.9982
Mawi-20191102	0.0004	0.9989
Mawi-20200901	0.0001	0.9995
Mendeley	0.0196	0.9275
Sanjose-20081016	0.0145	0.9793

El cuadro muestra que la entropía estimada difiere significativamente de las entropía reales calculadas en el cuadro 5.1. Además, notemos que no hay relación alguna entre la entropía estimada de una traza y su entropía real, es decir, la entropía estimada no corresponde a una fracción de la entropía real, lo cual sería una posibilidad ya que el algoritmo toma una muestra del stream de datos.

El problema con la extensión del HyperLogLog es que los heavy hitters (elementos más frecuentes) captados por la muestra son muy pocos. Dependiendo de cómo se definan los heavy hitters, en la muestra tomada por el sketch están en las decenas, mientras que para el HyperLogLog con CountMin-CU los heavy hitters están en los miles. Agrandar el HyperLogLog no mejora significativamente la cantidad de heavy hitters en la muestra. Además, los elementos captados por la muestra no estan distribuidos de manera equitativa, lo que se aprecia en los histogramas, lo cual afecta al cálculo de la entropía.

Así, la extensión del HyperLogLog descrita en el paper[4] no sirve para estimar la entropía de las trazas de red. Por último, se utilizó la extensión del algoritmo HyperLogLog para estimar la contribución a la entropía de los elementos descartados de las colas de prioridad del sketch CountMin-CU. Se utilizarón los parámetros d=8, w=16834 y p=8. Se calculó el error relativo de la contribución estimada de esta manera con la contribución real de los elementos descartados. La contribución estimada de esta manera se muestra en el cuadro 5.21.

El cuadro muestra que el error relativo es casi igual a 1, por lo que el algoritmo tampoco sirve para estimar la contribución de los elementos descartados por las colas de prioridad.

5.6. Tiempo de ejecución con paralelismo

Para la medición del tiempo, se ejecutó el programa entregándole de antemano la cantidad de líneas del archivo, para no ejecutar el contador automático de líneas, y además se leyó de a una línea del archivo a memoria a la vez. El tiempo se midió en segundos, llamando al comando time de Linux.

A continuación se presentan los resultados de la ejecución paralela (con 4 hebras) de la implementación con múltiples CountMin-CU para diferentes configuraciones. Los resultados se muestran en el cuadro 5.22.

Gráficamente, podemos representar los tiempos de ejecución secuencial y

Cuadro 5.21: Contribución estimada insertando elementos descartados a extensión del sketch HyperLogLog, con parámetros d=8, w=16384 y p=8

Traza	Contribución estimada	Error relativo
Chicago-20080319	0.0011	0.9961
Chicago-20080515	0.0010	0.9953
Chicago-20110608	0.0005	0.9971
Chicago-20150219	0.0009	0.9935
Chicago-20160121	0.0014	0.9765
Mawi-20161201	4.2e-05	0.9996
Mawi-20171101	5.1e-05	0.9992
Mawi-20181201	4.9e-05	0.9995
Mawi-20191102	4.8e-05	0.9996
Mawi-20200901	4.6e-05	0.9994
Mendeley	0.0019	0.9806
Sanjose-20081016	0.0006	0.9972

paralelo para la implementación con múltiples sketches CountMin-CU como se muestra en la imagen 5.18. La clave en el eje x es la misma utilizada en el cuadro 5.1.

De donde vemos que el tiempo de ejecución no cambia para las distintas configuraciones. Además, el tiempo de ejecución se redujo en aproximadamente un 50 % respecto de la ejecución secuencial.

Calculamos la aceleración y eficiencia de la ejecución paralela para cada archivo y configuración, las cuales se muestran en los cuadros 5.23 y 5.24.

Naturalmente, vemos que las aceleraciones y eficiencias no varían entre las configuraciones, ya que los tiempos de ejecución secuencial y paralelo tampoco variaron entre estas.

Es importante notar que el uso de paralelismo (y de hebras en general, incluso si se ejecutan de manera secuencial) induce un pequeño error en la estimación de la contribución a la entropía de los elementos más frecuentes, el cual no afecta de manera notoria al error relativo. Este error ocurre al alterarse el orden en que se introducen las frecuencias de las elementos en las colas de prioridad, ya que puede ocurrir que múltiples hebras en un momento dado hagan uso de la misma cola de prioridad, la cual es de tamaño fijo, por lo que puede ocurrir que la cola esté llena, y el reemplazo del elemento menos frecuente dependa del orden en que se ejecuten las hebras.

Cuadro 5.22: Tiempo de ejecución paralelo con múltiples sketches CountMin-CU

Traza	Tiempo p=6	Tiempo p=7	Tiempo p=8
Chicago-20080319	2.16	2.19	2.17
Chicago-20080515	7.25	6.84	6.58
Chicago-20110608	14.48	14.64	14.57
Chicago-20150219	8.47	8.54	8.46
Chicago-20160121	16.52	16.63	16.27
Mawi-20161201	51.72	50.40	49.40
Mawi-20171101	60.12	60.91	60.50
Mawi-20181201	39.92	40.52	39.88
Mawi-20191102	33.00	33.17	32.89
Mawi-20200901	63.20	64.14	63.42
Mendeley	1.50	1.57	1.50
Sanjose-20081016	11.17	11.35	11.20

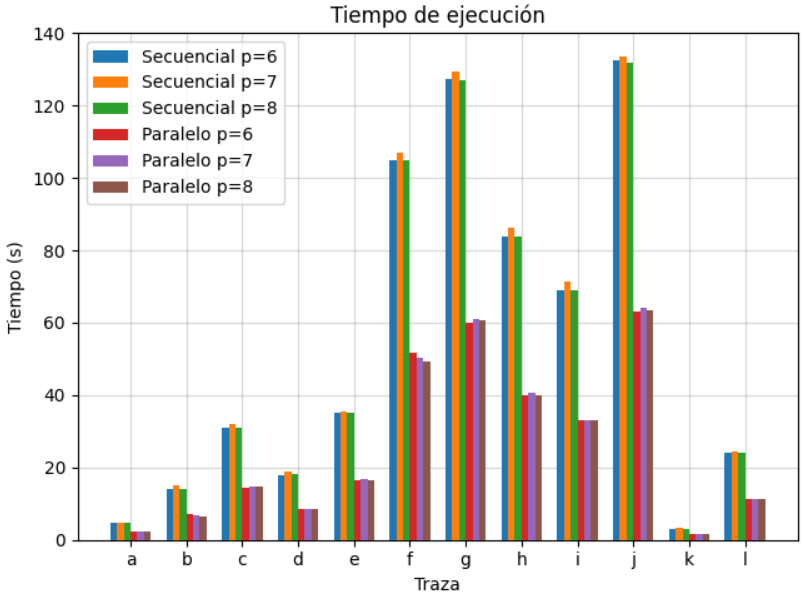


Figura 5.18: Tiempo de ejecución secuencial y paralelo

Cuadro 5.23: Aceleración (S) para cada archivo y configuración

Traza	S p=6	S p=7	S p=8
Chicago-20080319	2.1805	2.1780	2.1889
Chicago-20080515	1.9531	2.1812	2.1534
Chicago-20110608	2.1374	2.1762	2.1249
Chicago-20150219	2.1227	2.1920	2.1264
Chicago-20160121	2.1180	2.1371	2.1493
Mawi-20161201	2.0307	2.1192	2.1208
Mawi-20171101	2.1175	2.1236	2.1014
Mawi-20181201	2.0979	2.1310	2.0987
Mawi-20191102	2.0887	2.1546	2.0963
Mawi-20200901	2.0968	2.0807	2.0815
Mendeley	1.9733	2.0127	1.9666
Sanjose-20081016	2.1486	2.1603	2.1473

Cuadro 5.24: Eficiencia (E) para cada archivo y configuración

Traza	E p=6	E p=7	E p=8
Chicago-20080319	0.5451	0.5445	0.5472
Chicago-20080515	0.4882	0.5453	0.5383
Chicago-20110608	0.5343	0.5440	0.5312
Chicago-20150219	0.5306	0.5480	0.5316
Chicago-20160121	0.5295	0.5342	0.5373
Mawi-20161201	0.5076	0.5298	0.5302
Mawi-20171101	0.5293	0.5309	0.5253
Mawi-20181201	0.5244	0.5327	0.5246
Mawi-20191102	0.5221	0.5386	0.5240
Mawi-20200901	0.5242	0.5201	0.5203
Mendeley	0.4933	0.5031	0.4916
Sanjose-20081016	0.5371	0.5400	0.5368

Capítulo 6

Conclusiones

Se encontró que el algoritmo HyperLogLogLog no sirve como reemplazo de nuestra implementación del algoritmo HyperLogLog, con múltiples registros por celda, en el contexto de disminución del uso de memoria, ya que el uso de tablas hash afecta negativamente al uso de memoria. Además, el algoritmo HyperLogLogLog tiene un tiempo de ejecución demasiado grande y es cuadrático respecto al tamaño del sketch, ya que por cada elemento del stream insertado, se debe recorrer todo el sketch., lo cual limita significativamente el tamaño máximo del sketch que se puede utilizar.

Estimar la entropía mediante múltiples sketches CountMin-CU da los mismos resultados que utilizar solo un sketch CountMin-CU global, pero utiliza ligeramente más memoria, principalmente por el uso de locks, donde la cantidad de locks está dada por la cantidad de sketches CountMin-CU. Además, utilizar múltiples sketches CountMin-CU permite utilizar paralelismo con OpenMP, y se encontró que utilizar 4 hebras reduce el tiempo de ejecución en un 50 % respecto de la ejecución secuencial. El tiempo de ejecución secuencial para la implementación con múltiples sketches CountMin-CU es el mismo que el tiempo de ejecución de la implementación con un único sketch CountMin-CU global.

El algoritmo TowerSketch-CU permite estimar la entropía de las trazas de red pero sin ninguna mejora en precisión o uso de memoria respecto del algoritmo CountMin-CU, ya sea utilizando un sketch global o un sketch por celda del sketch HyperLogLog. Además, tiene un tiempo de ejecución un 50 % más lento que el sketch CountMin-CU.

La extensión del algoritmo HyperLogLog, que permite realizar un muestreo al stream de datos, permite, dependiendo del tamaño del sketch y del

tamaño del stream de datos, capturar una muestra representativa, lo cual se ve reflejado en el histograma de frecuencias, pero tiene problemas para captar los elementos de mayor frecuencia. Además, el algoritmo no sirve para calcular la contribución a la entropía de los elementos descartados de las colas de prioridad asociadas a un sketch CountMin-CU.

Ya que se consiguió mejorar significativamente el tiempo de ejecución del algoritmo utilizando múltiples sketches CountMin-CU mediante el uso de paralelismo, se recomienda utilizar una arquitectura con múltiples pipelines, o bien otro método de paralelismo soportado por el lenguaje de programación P4, implementando el algoritmo con múltiples sketches CountMin-CU.

Bibliografía

[1] I. G. C. A. Goyal, H. Daumé. Sketch algorithms for estimating point queries in nlp. *Proc. Joint Conf. Empirical Methods Natural Lang. Process. Comput. Natural Lang. Learn*, pages 1093–1103, 2012.

[2] D. S. D. Ding, M. Savi. Estimating logarithmic and exponential functions to track network traffic entropy in p4. *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, page 1–9, 2020.

[3] E. B.-H. Elie F. Kfoury, Jorge Crichigno. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *arXiv*, 2021.

[4] W. B. J. B. A. U. L. T. Evgeny Skvortsov, Jeff Wilhelm. Tracking audience statistics with hyperloglog. *Google LLC*, 2021.

[5] M. M. G. Cormode. Approximating data with the countmin sketch. *IEEE Softw*, 29:64–69, 2012.

[6] I. B. Hamid Mohamadi, Hamza Khan. ntcard: a streaming algorithm for cardinality estimation in genomics data. *Oxford Journals*, 33(9):1324–1330, 2017.

[7] Y. F. C. H. M. F. J. E. Soto, P. Ubisse. A high-throughput hardware accelerator for network entropy estimation using sketches. *IEEE Access*, 9:85823–85838, 2021.

[8] Z. L. T. Y. Y. Z. J. H. J. X. T. Z. Z. J. Y. Y. Kaicheng Y., Yuanpeng L. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. *IEEE Xplore*, pages 1–12, 2021.

[9] P. R. Karppa, M. Hyperlogloglog: Cardinality estimation with one log more. *arXiv preprint arXiv:2205.11327*, 2022.

[10] O. G. F. M. P. Flajolet, E. Fusy. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. *Proc. 13th Conf. Anal. Algorithms (AoA)*, 29:127–146, 2007.