

Hashing

Organización de Datos 1C2020

Definición

$$h: U \longrightarrow \{0, 1, \dots, m-1\}$$

Espacio de Claves

Espacio de direcciones

Motivación

- Nos permite trabajar con números en vez de strings, texto, imágenes, audio
- Algunas son muy rápidas y algunas son muy seguras
- Nos permiten partir los datos fácilmente
- La vida es mejor con funciones de hash

Función de hash

¿Qué propiedades nos gustaría que tenga?

Ejemplos

Algorithm 11: FNV Hashing

Data: s : string

Result: h : integer

```
1  $h = 14695981039346656037$ ;  
2 for  $c$  in  $s$  do  
3    $h = h * 1099511628211$ ;  
4    $h = h \text{ XOR } c$ ;  
5 return  $h$ ;
```

El valor del número h
depende del tamaño
del hash n

El valor del número
primo también
depende de n !

Ejemplos

Simplemente usa shifts y XOR. Funciona muy bien para LookUp en tablas hash

Algorithm 12: Jenkins Hashing

Data: s: string

Result: hash: integer

```
1 for  $hash=i=0; i < len; i++$  do  
2    $hash += s[i]$   
3    $hash += (hash << 10);$   
4    $hash \text{ xor} = (hash >> 6);$   
5  $hash += (hash << 3);$   
6  $hash \text{ xor} = (hash >> 11);$   
7  $hash += (hash << 15);$   
8 return hash;
```

Ejemplos

Algorithm 13: Pearson Hashing

Data: s : string

Result: hash: integer

```
1  $h = 0$ ;  
2 for  $c$  in  $s$  do  
3   |    $\text{index} = h \text{ xor } c$ ;  
4   |    $h = T[\text{index}]$ ;  
5 return  $h$ 
```

T : Tabla con
permutación aleatoria
de los números 0..255

Funciones hash no criptográficas

Son aquellas funciones que cumplen las siguientes propiedades:

- Deben ser muy eficiente calcular $h(x)$
- Debe producir la menor cantidad de colisiones posibles

Ejemplos: FNV, Jenkins, Murmur, Pearson

Funciones hash criptográficas

Son aquellas funciones que cumplen las siguientes propiedades:

- Dado $h(x)$ tiene que ser muy difícil hallar x
- Tiene que ser muy difícil hallar x e y tal que $h(x) = h(y)$
- La función tiene que producir la menor cantidad de colisiones posibles
- Debe producir **efecto avalancha**

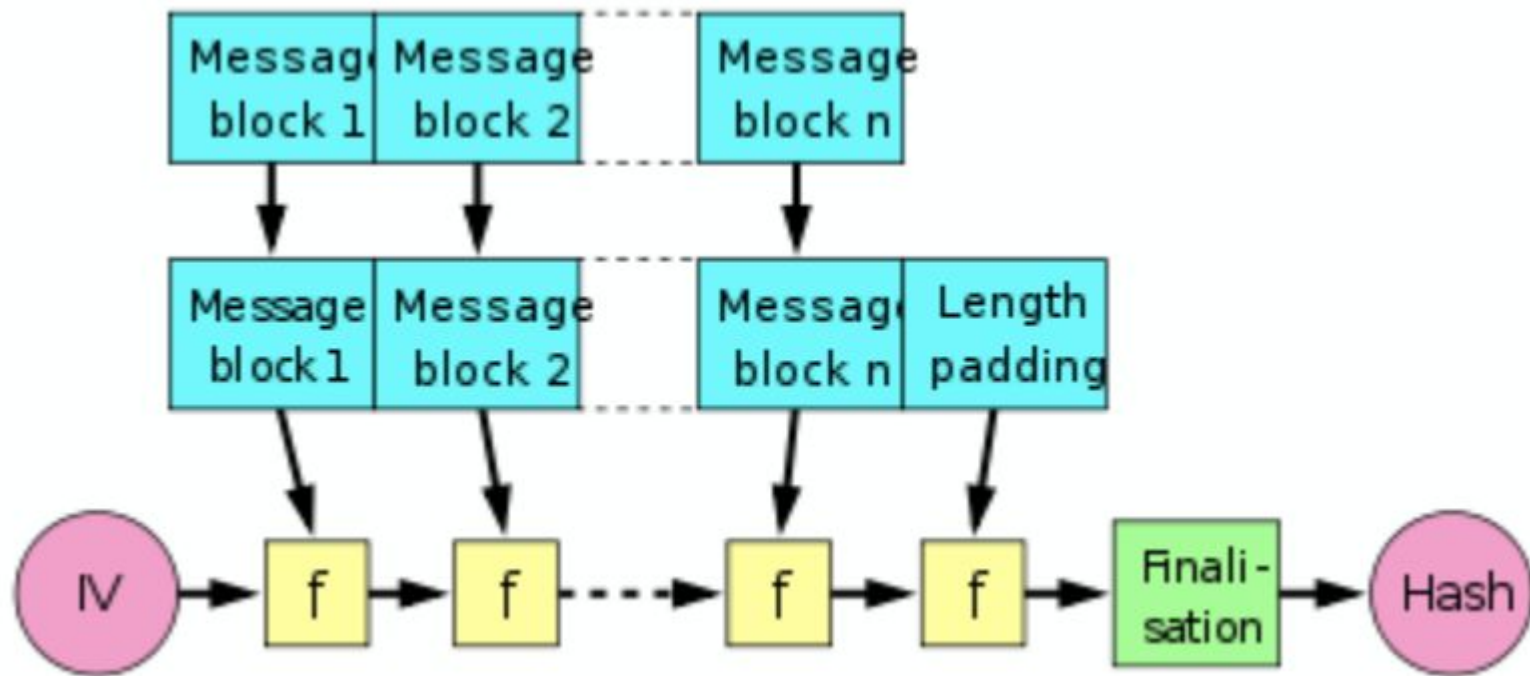
Ejemplos: MD5, SHA-256/512, Blake2

Funciones de Hashing criptográficas (CHF)

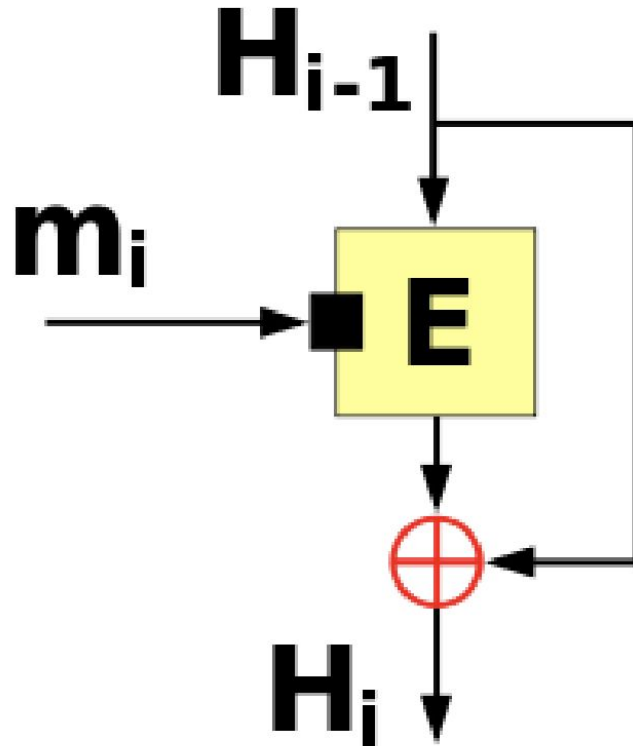
Muchos usos:

- Firmas para archivos (integridad e identificación)
- Almacenado de contraseñas (+*salt*)
- Bitcoin

SHA-256: Merkle-Damgård



SHA-256: Davies-Meyer



En SHA-256, el algoritmo de encripción que se utiliza es *Shacal-2*

Repaso

- ¿Es fácil o difícil encontrar colisiones de $h_{\text{criptografica}}(h_{\text{generica}}(x))$?
- ¿Conviene usar una criptográfica para una tabla hash?

Hashing universal

Def: H es una familia de funciones de hashing universal si se cumple que:

$$\forall x, y \in U, h \in \mathcal{H}, x \neq y$$

$$P[h(x) = h(y)] \leq \frac{1}{m}$$

Hashing universal

Construcción para valores numéricos

$$h \in \mathcal{H} \Rightarrow h(x) = (a * x + b \pmod{p}) \pmod{m}$$

m: espacio de direcciones

p: número primo $\geq m$

a: $\{1, 2, \dots, p-1\}$ b: $\{0, 1, \dots, p-1\}$

Hashing universal

Construcción para claves de long. fija

$$h \in \mathcal{H} \Rightarrow h(x) = \sum_{i=0}^r (a_i * x_i \pmod{p}) \pmod{m}$$

m: espacio de direcciones

p: número primo $\geq m$

a_i : $\{0, 1, 2, \dots, p-1\}$

Hashing universal

Construcción para claves de long. variable

$$h \in \mathcal{H} \Rightarrow, h(x) = h_{int}((\sum_{i=1}^l x_i * a^i) \mod p)$$

m: espacio de direcciones

p: número primo muy grande

a: $\{0,1,2\dots p-1\}$

Hashing perfecto

- Hashing donde no hay colisiones
- Garantiza $O(1)$ en búsqueda
- ¿Cómo lo construimos?

FKS

- $O(1)$ en búsqueda
- $O(n)$ en espacio ($\leq 2n$)
- Resolvemos colisiones buscando funciones de hash que diferencien las claves

FKS

Intentemos almacenar $\{1, 5, 6, 9, 12, 13\} \gg$ mínimo $m = 6$.

Elegimos $k = 7$ (primo cercano a m).

Usamos, por ejemplo, $h(x) = x \bmod 7$

Quando hay colisiones de tamaño m_i , elegimos un k_i primo más cercano a m_i^2 , y elegimos h_i tal que no colisione.

k	claves	hash2	segundas tablas
0			
1	1	$x \bmod 1$	$\{1\}$
2	9	$x \bmod 1$	$\{9\}$
3			
4			
5	5, 12	$x \bmod 5$	$\{5, -, 12, -, -\}$
6	6, 13	$2x \bmod 5$	$\{-, 13, 6, -, -\}$

Hashing perfecto y mínimo

- $O(1)$ en búsqueda
- Buscamos acercarnos al óptimo en cuanto a espacio (no queremos dejar slots vacíos en la tabla)

HDC

- Necesitamos una familia del estilo $h(i,x)$ con $i = 0..n-1$ (n espacio de claves)
- Todos los slots de la tabla quedarán ocupados por una clave

HDC

Tenemos m claves a almacenar, y tenemos un hash *parametrizable* $h(i, x)$

Hasheamos todas las claves con $h(o, x) \Rightarrow$ va a haber colisiones

Marcamos en un **bitmap** las posiciones ocupadas por una sola clave.

Partiendo del bucket con más colisiones, aumentamos i hasta que las claves se ubiquen en espacios vacíos, según el bitmap. Repetimos, nunca decrementando i , y actualizando el bitmap.

En una tabla G anotamos los valores i que se usó para resolver x ; si con $h(o, x)$ no hubo colisión almacenamos $-h(o, x)$.

Repaso de tablas de hash

- Las tablas de hash se diferencian en cómo manejan los **sinónimos**
- En Algoritmos 2 vieron al menos dos formas:
 - sondeo lineal (forma de *hashing cerrado*)
 - encadenamiento de sinónimos (forma de *hashing abierto*)
- Todas las tablas rehashean; hay noción de **factor de carga**

Variaciones sobre hashing cerrado (dir. abierto)

- Hashing cerrado es bueno por el comportamiento de **caché**
- Resuelve colisiones según política o **secuencia de sondeo**
 - p. ej. $H+1$, $H+2$, $H+3$,... (sondeo lineal)
- El problema del sondeo lineal es la generación de grupos o **clusters** (problema de **clustering primario**)
 - Llena espacios contiguos y degrada la inserción de claves con distinto hash

⇒ Vamos a variar la secuencia de sondeo

Variaciones sobre hashing cerrado (dir. abierto)

Algunas formas de sondeo:

- Lineal: $H+1$, $H+2$, $H+3$, ...
- Cuadrático: $H+1^2$, $H+2^2$, $H+3^2$, ...
- Hashing doble: $H+1H'$, $H+2H'$, $H+3H'$

$H \equiv \text{hash}(\text{clave})$; $H' \equiv \text{hash_2}(\text{clave})$
generalmente hash y hash_2 son de la misma familia

Cuckoo hashing

- Tenemos dos tablas con funciones de hash asociadas
- Un dato siempre está en $h_1(x)$ o $h_2(x)$ (**tiempo constante**)
- Ante una colisión, la clave nueva echa la vieja a la otra tabla

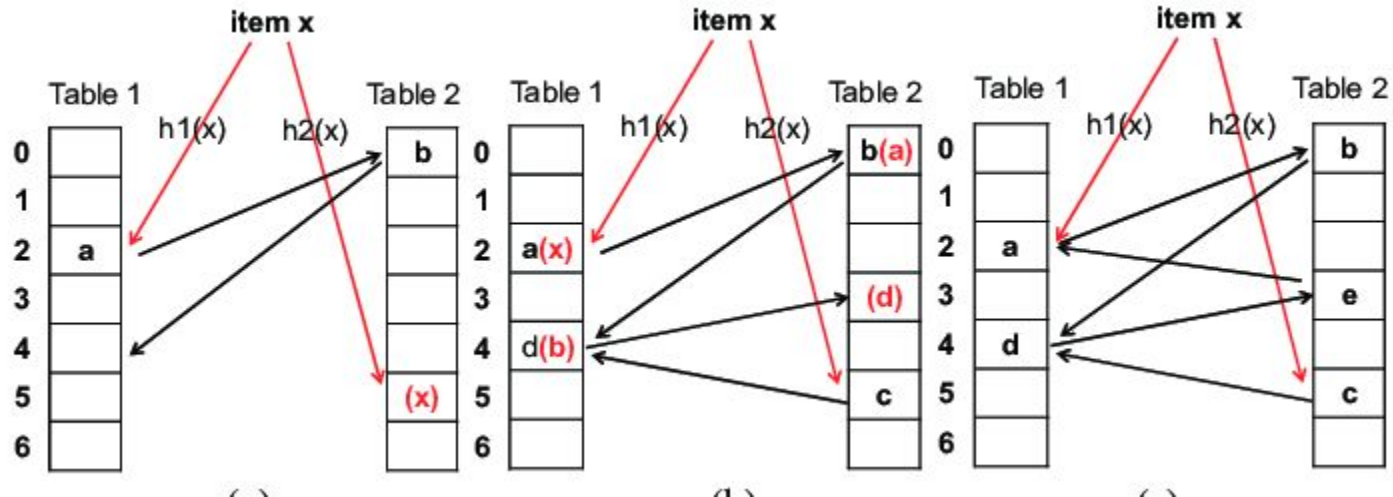


This [Eurasian reed warbler](#) is raising a common cuckoo.

Cuckoo hashing

```
function lookup( $x$ )  
    return  $T_1[h_1(x)] = x \vee T_2[h_2(x)] = x$   
end
```

Cuckoo hashing



Cuckoo hashing

```
procedure insert( $x$ )  
  if lookup( $x$ ) then return  
  loop MaxLoop times  
    if  $T_1[h_1(x)] = \perp$  then {  $T_1[h_1(x)] \leftarrow x$ ; return }  
     $x \leftrightarrow T_1[h_1(x)]$   
    if  $T_2[h_2(x)] = \perp$  then {  $T_2[h_2(x)] \leftarrow x$ ; return }  
     $x \leftrightarrow T_2[h_2(x)]$   
  end loop  
  rehash(); insert( $x$ )  
end
```

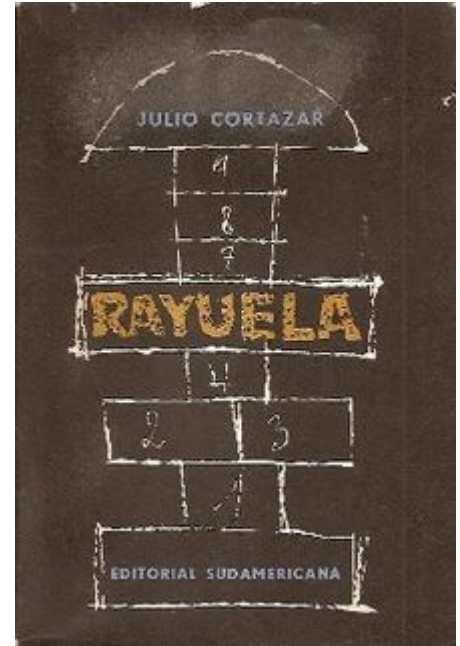
Nota: \leftrightarrow es swap, \perp es
“vacío”

Cuckoo hashing recargado

- En principio, podemos juntar las dos tablas en una
 - Podríamos tener más de 2 funciones de hash
- Podemos usar un *stash* para guardar los datos problemáticos y no rehashear

Hopscotch hashing

- Queremos evitar rehashear tanto (comparado con Cuckoo)
- Queremos evitar la mezcla de grupos de sinónimos que tiene el sondeo lineal \Rightarrow concepto de **vecindario**



Hopscotch hashing

- Cada posición tiene asociado un vecindario de tamaño V
- Si la posición $h(x)$ está ocupada, sólo puede ubicarse entre $[h(x), h(x)+V-1]$

Hasta ahora nada nuevo: los vecindarios se **solapan**

Hopscotch hashing

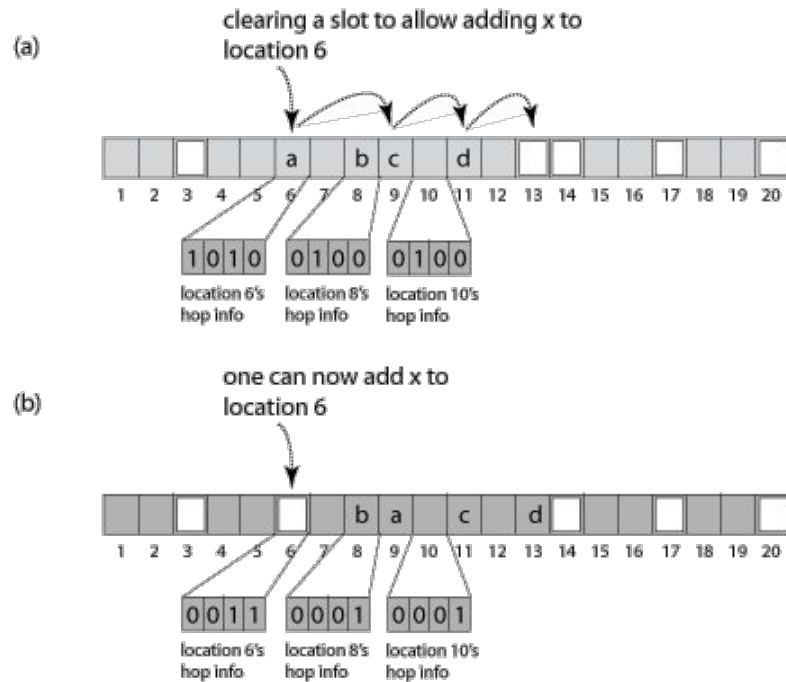
- Mantenemos un bitmap que nos dice cuáles de las siguientes $V-1$ posiciones tiene elementos que hashcan a $h(x)$.
 - Nos evitamos revisar elementos que no son sinónimos de x
- ¿Qué pasa si el vecindario de $h(x)$ se llena con no-sinónimos de x ?
 - Vamos a intentar reordenar las cosas
 - Si no se puede, rehashamos

Hopscotch hashing

Qué pasa cuando queremos insertar y $[h(x), h(x)+V-1]$ está todo ocupado pero el bitmap de $h(x)$ tiene ceros:

1. Buscamos la primera posición vacía
2. Vamos hacia atrás viendo si podemos mover algún dato a esa posición, repetimos las veces que sea necesario
3. Si no se puede, rehasheamos

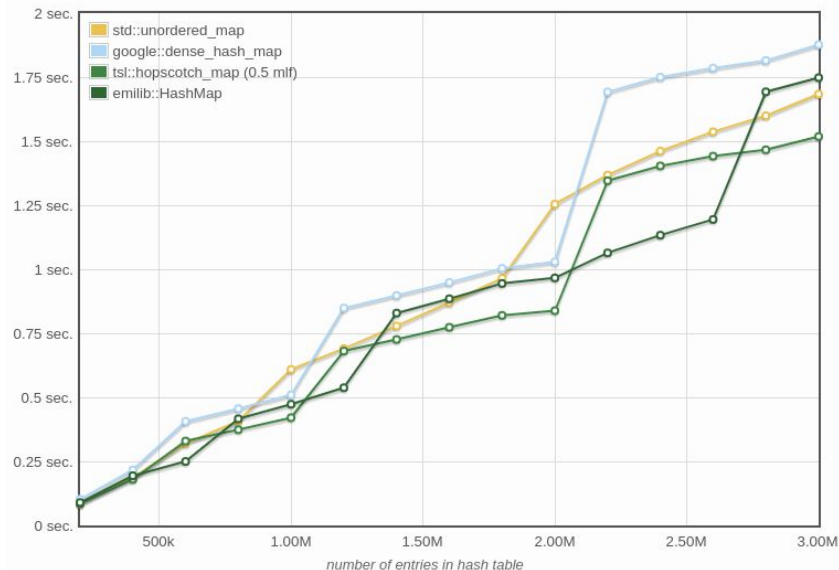
Hopscotch hashing



Hopscotch hashing

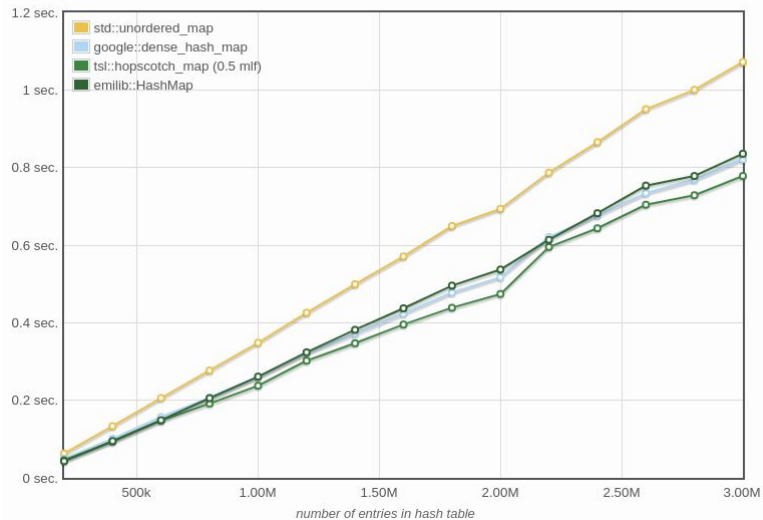
Inserts: execution time (strings)

For each entry in the range $[0, \text{nb_entries})$, we generate a string as key and insert it with the value 1.



Inserts with reserve: execution time (strings)

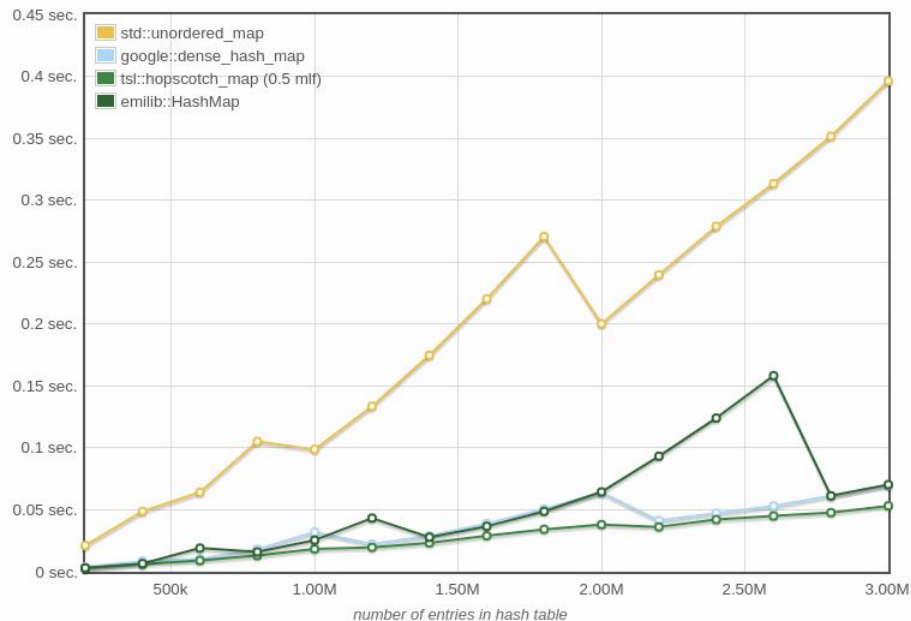
Same as the inserts test but the reserve method of the hash map is called beforehand to avoid any rehash during the insertion. It provides a fair comparison even if the growth factor of each hash map is different.



Hopscotch hashing

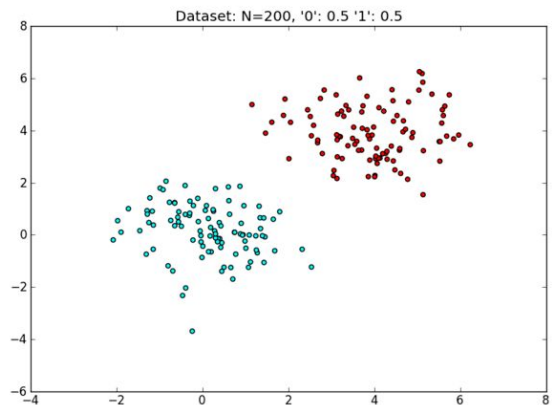
Random full reads misses: execution time (integers)

Before the test, we insert `nb_entries` elements in the same way as in the random full inserts test. We then generate another vector of `nb_entries` random elements different from the inserted elements and we try to search for these unknown elements in the hash map.



Feature Hashing

- Vamos a aplicar funciones de hash para **reducir las dimensiones** de nuestro set de datos
- La idea es mantener las diferencias (distancias) entre los puntos



Interludio: Lema de Johnson-Lindenstrauss

Si tenemos n puntos en \mathbb{R}^N , y definimos un error ϵ con $0 < \epsilon < 1$:

$$\exists k = O(\ln n) \wedge k > 24\epsilon^{-2} \ln n, \quad f : \mathbb{R}^N \rightarrow \mathbb{R}^k.$$

$$(1 - \epsilon) \|u - v\|^2 \leq \|f(u) - f(v)\|^2 \leq (1 + \epsilon) \|u - v\|^2$$

(para todo u, v)

Interludio: Lema de Johnson-Lindenstrauss - Proyecciones Aleatorias

Una matriz de proyección aleatoria puede ser una matriz A de tamaño $\mathbf{R}^{N \times k}$ tal que A_{ij} = valor aleatorio normal.

Proyectamos los datos:

$$\underset{n \times N}{X} \underset{N \times k}{A} = \underset{n \times k}{\hat{X}}$$

Esta nueva matriz preserva aproximadamente las distancias entre los datos.

Feature hashing

- Vamos a analizar otra proyección: una matriz donde sólo un elemento por fila es 1.
- Es decir, la nueva dimensión i está compuesta por la suma de las dimensiones originales j que hashean a i .

Si los datos tienen 7 dimensiones y usamos h : $h(x) = ((3x + 1)) \mod 5) \mod 4$

es lo mismo que multiplicar mis datos **por**:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Feature hashing

Podemos escribir x pasado por feature hashing como:

$$\phi^{(h)}(x)$$

Las componentes del nuevo vector son:

$$\phi_i^{(h)}(x) = \sum_{j:h(j)=i} x_j$$

Feature hashing - función signo

Si agregamos **otra función hash** que actúa de **signo** mejora los resultados:

$$\phi_i^{(h, \xi)}(x) = \sum_{j: h(j)=i} \xi(j) x_j$$

$$\xi : X \rightarrow \{-1, 1\}$$

y esta proyección **sí** cumple el lema JL.

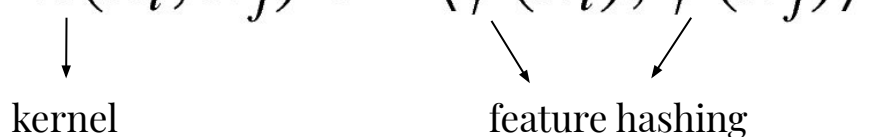
The Hashing Trick

- El **producto interno** entre dos datos nos da un indicio de **similaridad**
 - e.g. >0 son similares, aprox. 0 son independientes, <0 son opuestos
- A veces es caro (tiempo o memoria) calcularlo con todos los features
- Si tan solo tuviésemos menos features con la misma estructura...

The Hashing Trick

- Podemos aplicar feature hashing y calcular el producto interno en ese espacio
- En un solo paso:

$$k(x_i, x_j) := \langle \phi(x_i), \phi(x_j) \rangle$$



kernel feature hashing

Aplicaciones de THT - detección de spam

- Los mails spam generalmente usan un lenguaje similar
- Podemos medir similaridad entre mails comparando las palabras que usan
- ¡Pero esto es muy lento!

Bibliografía

Introduction to Algorithms. Thomas Cormen

Universal classes of hash functions. Carter y Wegman

Ladrillo de la materia

Bibliografía

Cuckoo Hashing, Pagh R. & Rodler F.

Hopscotch Hashing. Herlihy et al.

Benchmark of Hashmap Implementations

Feature Hashing for Large Scale Multitask Learning,
Weinberger et al.

Bibliografía

An elementary proof of a theorem of Johnson and Lindenstrauss, Dasgupta & Gupta

Extensions of Lipschitz maps into a Hilbert space, Johnson & Lindenstrauss