

MPEI

Geração de números aleatórios

Motivação (exemplos)

- Gerar strings “aleatórias” em que:
 - comprimento assume valores entre 1 e 10 e tendo cada comprimento a mesma probabilidade
 - o carácter em cada posição é uma das letras minúsculas ou maiúsculas do alfabeto português e tendo todas as letras a mesma probabilidade
- Gerar strings em que quer as letras quer o comprimento assumem distribuições mais próximas da realidade
 - Comprimento seguindo uma distribuição Normal com média e variância estimada de um conjunto de textos
 - As letras seguem a distribuição para o Português
 - Que vimos numa aula anterior

Geradores

- Para situações como as do exemplo, necessitamos de resolver o problema de gerar, ou simular, **vectores de números aleatórios tendo uma determinada distribuição**
- Nos primeiros tempos da simulação utilizavam-se métodos mecânicos para obter valores aleatórios: Moedas, dados, roletas, cartas
- Mais tarde utilizaram-se propriedades de dispositivos e elementos
 - Exemplos (atuais):
 - www.fourmilab.ch/hotbits (decaimento do Césio-137)
 - www.random.org/integers (ruído atmosférico)
- Na área da Informática e outras, estes métodos foram substituídos por algoritmos que se podem implementar facilmente em computador , os **Geradores de números pseudo-aleatórios**
 - Capazes de criar sequências numéricas com propriedades próximas de sequências aleatórias
 - São algoritmos determinísticos, pelo que é usual designar os números gerados por “pseudo-aleatórios”

Abordagens principais

- Gerar directamente
- Gerar número “aleatório” de uma distribuição uniforme (contínua) e transformar ...
 - Neste caso, torna-se necessário ser capaz de gerar variáveis aleatórias com a distribuição uniforme
 - Em geral distribuída entre 0 e 1
 - É a abordagem comum

Geração de variáveis aleatórias com distribuição uniforme entre 0 e 1

Algoritmos congruenciais

- Os métodos mais comuns para gerar sequência pseudo-aleatórias usam os chamados *linear congruential generators - LCG* (algoritmo congruencial linear)
- Este geradores geram uma sequência de números através da **fórmula recursiva**

$$X_{i+1} = (aX_i + c) \bmod m$$

- Com X_0 sendo a “semente” (seed) e a, c, m (todos inteiros positivos) designados de multiplicador, incremento e módulo, respetivamente
- Quando $c = 0$ o algoritmo designa-se por **congruencial multiplicativo**

Algoritmos congruenciais

- Como X_i pode apenas assumir os valores $\{0, 1, \dots, m-1\}$, os números

$$U_i = \frac{X_i}{m}$$

são designados por número pseudo-aleatórios e constituem uma aproximação a uma sequência de variáveis aleatórias uniformemente distribuídas

Processo de cálculo em detalhe

1. Escolher os valores de a , c e m
2. Escolher a semente X_0 (tal que $1 \leq X_0 \leq m$)
3. Calcular o próximo número aleatório usando a expressão $X_1 = (aX_0 + c) \bmod m$
4. Substituir X_0 por X_1 e voltar ao ponto anterior

Exemplo

- Fazendo $a=9$, $c=1$, $m=17$ e $X_0 = 7$

n	x_n	$y=9x_n+1$	$y \text{mod } 17$	$x_{n+1}/17$
0	$X_0=7$	$9*7+1=64$	13	$13/17 = 0.7647$
1	$X_1=13$	118	16	$16/17 = 0.9412$
2	$X_2=16$	145	9	0.5294
3	$X_3=9$	82	14	0.8235
4	$X_4=14$	127	8	0.4706

números pseudo-aleatórios inteiros entre 0 e 16 ($=17-1$)

números pseudo-aleatórios inteiros entre 0 e 1

Como escolher os parâmetros ?

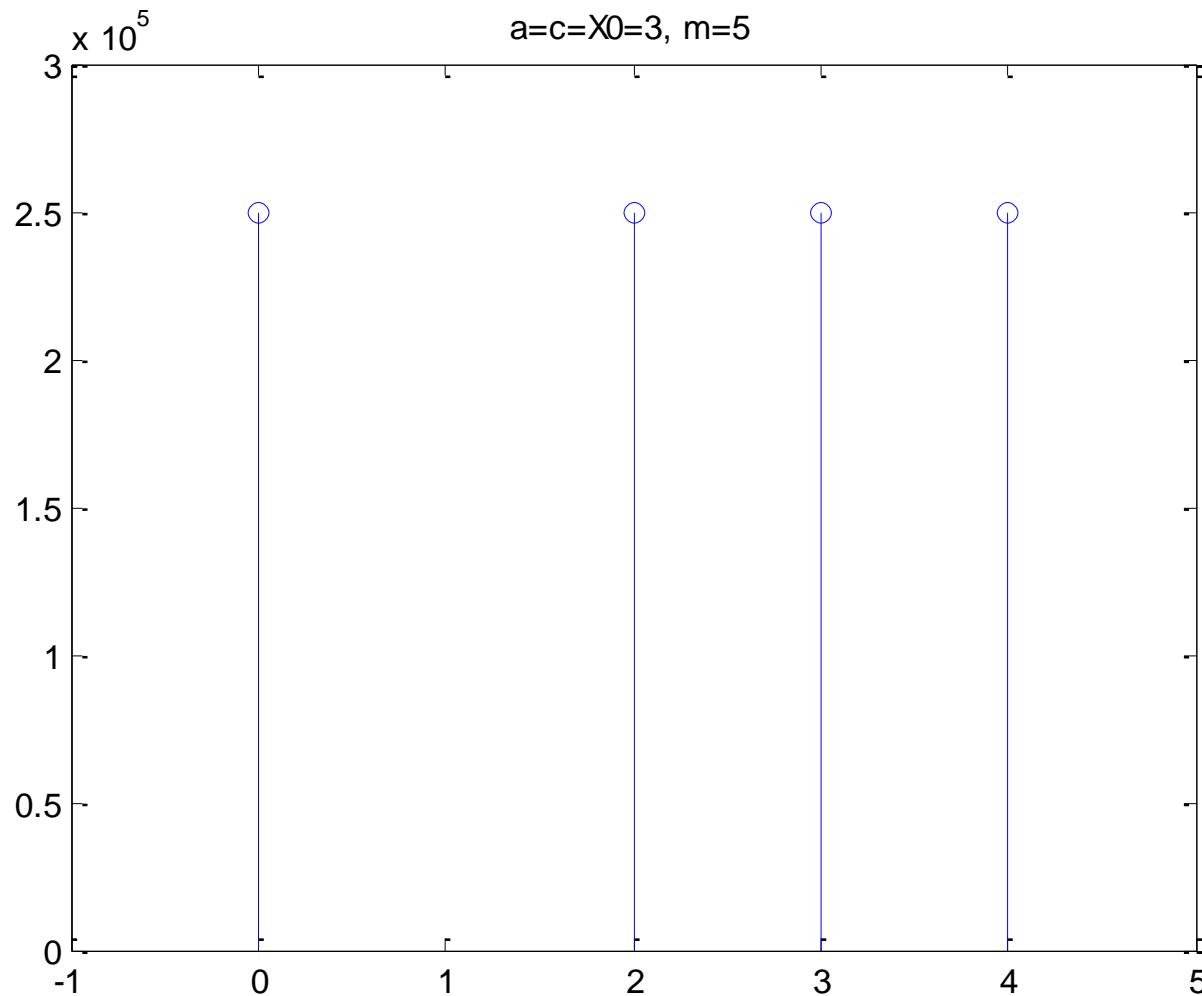
- A ciclo de repetição da sequência é no máximo m
- Será, portanto, periódica com um período que não excede m
- Mas pode ser muito pior
 - Exemplo: $a=c= X_0=3$ e $m=5$ gera a sequência $\{3,2,4,0,3 \dots\}$ com período 4
- Apenas algumas combinações de parâmetros produzem resultados satisfatórios
 - Exemplo: Usar $m = 2^{31} - 1$ e $a = 7^5$ em computadores de 32 bits

Demo Matlab

- Exemplo: $a=c= X_0=3$ e $m=5$ gera a sequência $\{3,2,4,0,3 \dots\}$

```
function U=lcg(X0,a,c,m, N)
U=zeros(1,N);
U(1)=X0;
for i=2:N
    U(i) = rem(a*U(i-1)+c, m);
end
```

Resultados - histograma



Outros algoritmos congruenciais

- Uma generalização que se pode fazer do algoritmo congruencial multiplicativo é basear o cálculo do novo valor numa combinação linear das k amostras anteriores
- Um exemplo deste tipo baseia-se na sequência de **Fibonacci**

$$x_i = x_{i-1} + x_{i-2}, \quad x_1 = 1, x_0 = 0$$

- Como a utilização directa não dá bons resultados, usa-se

$$x_i = (x_{i-j} + x_{i-k}) \bmod m$$

- Para $j=31, k=63, m=2^{64}$ temos período de 2^{124}

Outros algoritmos congruenciais

- Outra estratégia: **combinar os resultados obtidos com dois geradores congruenciais** que, com a escolha conveniente dos parâmetros, vai permitir maiores períodos
 - Conhecida por **Combined Multiple Recursive Generator**
- Na implementação em Matlab consiste em:

$$x_{1,n} = (14033580x_{1,n-2} - 810728x_{1,n-3}) \bmod m_1$$

$$x_{2,n} = (527612x_{2,n-1} - 1370589x_{2,n-3}) \bmod m_2$$

- Sendo a saída

$$z_n \equiv (x_{1,n} - x_{2,n}) \bmod m_1$$

$$u_n = \begin{cases} z_n/(m_1 + 1) & , z_n < 0 \\ m_1/(m_1 + 1) & , z_n = 0 \end{cases}$$

Outros geradores

- **FSR** – Feedback Shift Register
 - Relacionados com os geradores recursivos anteriores
 - A **formula recursiva é aplicada a bits**
 - Conjuntos de k bits representam inteiros
 - A formula de recursão é realizada recorrendo a um **Shift Register**
 - Vetor de bits que pode ser deslocado para a esquerda um bit de cada vez
 - Feita num computador recorrendo aos registos internos e programação em linguagem máquina
- Mersenne Twister
 - Desenvolvido para resolver problemas de uniformidade do FSR
 - Apresenta um período extraordinário de $2^{19937} - 1$
 - Informação em:
 - <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

Outros geradores

- Para além destes geradores, outras classes foram propostas por forma a obter períodos mais longos e melhor aproximação à distribuição uniforme
- A biblioteca NAG, por exemplo, inclui vários:

Pseudorandom Numbers
2.1.1 NAG Basic Generator
2.1.2 Wichmann–Hill I Generator
2.1.3 Wichmann–Hill II Generator
2.1.4 Mersenne Twister Generator
2.1.5 ACORN Generator
2.1.6 L'Ecuyer MRG32k3a Combined Recursive Generator . . .

Outros geradores - Exemplo

- Wichman-Hill I
- Usa uma combinação de 4 LCGs

This series of Wichmann–Hill base generators (see Maclaren (1989)) use a combination of four linear congruential generators and has the form:

$$\begin{aligned} w_i &= a_1 w_{i-1} \bmod m_1 \\ x_i &= a_2 x_{i-1} \bmod m_2 \\ y_i &= a_3 y_{i-1} \bmod m_3 \\ z_i &= a_4 z_{i-1} \bmod m_4 \\ u_i &= \left(\frac{w_i}{m_1} + \frac{x_i}{m_2} + \frac{y_i}{m_3} + \frac{z_i}{m_4} \right) \bmod 1, \end{aligned} \tag{1}$$

where the u_i , for $i = 1, 2, \dots$, form the required sequence. The NAG Library implementation includes 273 sets of parameters, a_j, m_j , for $j = 1, 2, 3, 4$, to choose from.

Na prática...

- A maioria das linguagens de computador disponibilizam geradores de números pseudo-aleatórios
 - Em geral o utilizador apenas fornece o valor da “semente”
- Java
 - Classe Random
 - Random rnd = new Random()
 - rnd.nextDouble()

Matlab

- A geração de números (pseudo-)aleatórios no Matlab baseia-se na geração de números uniformemente distribuídos no intervalo (0, 1) por um algoritmo similar aos anteriormente descritos, usando o comando `rand()`
- Por defeito `rand()` utiliza o algoritmo Mersenne twister
 - Mas permite que se altere, usando `rng()`

rng

- $s = \text{rng}$
- $s =$

struct with fields:

Type: 'twister' %% algoritmo por defeito

Seed: 0

State: [625×1 uint32]

rng

- **rng(seed, type)**

Type define o tipo de algoritmo usado e pode ser:

nome	descrição	state
'twister'	Mersenne Twister	625x1 uint32
'combRecursive'	Alg. multiplo recursivo	12x1 uint32
'multFibonacci'	Alg. Fibonacci multiplicativo com atraso	130x1 uint64
'v5uniform'	Gerador uniforme do MATLAB® 5.0	35x1 double
'v5normal'	Gerador normal do MATLAB 5.0	2x1 double
'v4'	Gerador do MATLAB 4.0	1 uint=seed

rand

```
>> rand % generate a uniform random number  
0.0196  
>> rand % generate another uniform random number  
0.823  
>> rand(1,4) % generate a uniform random vector  
0.5252 0.2026 0.6721 0.8381  
rand('state',1234) % set the seed to 1234  
>> rand % generate a uniform random number  
0.6104  
rand('state',1234) % reset the seed to 1234  
>> rand % the previous outcome is repeated  
0.6104
```

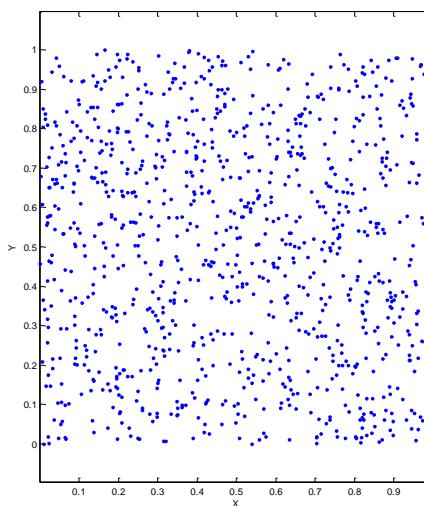
Demonstração do uso de rand()

N=1000

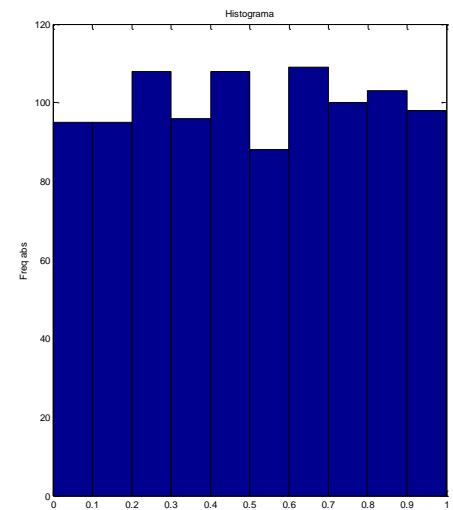
```
X = rand(1, N); Y= rand(1,N);
```

```
subplot(121), plot(X,Y,'.')  
axis equal
```

```
xlabel('X')  
ylabel('Y')
```



```
subplot(122), hist(X)  
title('Histograma');  
xlabel('X')  
ylabel('Freq abs');
```



Transformações

Transformações simples

- Aplicando a de **transformação linear** $Y=a U + b$ é simples obter variáveis com distribuição uniforme num intervalo
 - ex: $Y=2 U + 1$ permite intervalo [1, 3]
- A aplicação da transformação linear seguida da conversão para inteiros permite obter, por exemplo, uma simulação de lançamentos de um dado (**uma gama de números inteiros**)
 - Em versões mais recentes do Matlab existe mesmo a função `randi()`

Exemplos em Matlab

% geração de n resultados do lançamento de uma moeda

```
function Y=moeda(n)
```

```
if nargin ==0
```

```
    n=1;
```

```
end
```

```
z=round(rand(1,n));
```

```
Y(1:n)='C'; % CARA
```

```
Y(find(z==0))='R'; % COROA
```

% usando

```
moeda(10)
```

Exemplos em Matlab

% n resultados do lançamento de um dado

```
function Y=dado(n)
```

```
if nargin==0
```

```
    n=1;
```

```
end
```

```
Y=floor(rand(1,n)*6)+1; %% ou randi(6,1,n)
```

dado → 5

dado(10) → 3 1 4 5 6 3 4 3 2 4

Métodos Genéricos para gerar variáveis aleatórias com distribuições não uniformes

Métodos

- Números aleatórios com outras distribuições podem ser obtidos das sequências com distribuição uniforme através de:
 - Métodos de transformação
 - Métodos de rejeição
 - Procura em tabelas

Método da Transformação (Inversa)

- Para uma v.a. contínua, se a função de distribuição acumulada é $F(x)$ então para uma variável U com distribuição uniforme em $(0,1)$

$$X = F^{-1}(U) \text{ tem por função distrib. acum. } F(x)$$

- Este método é apenas eficiente num conjunto pequeno de casos (ex: distribuição exponencial)
- Também não é possível ou é difícil determinar a inversa de muitas distribuições

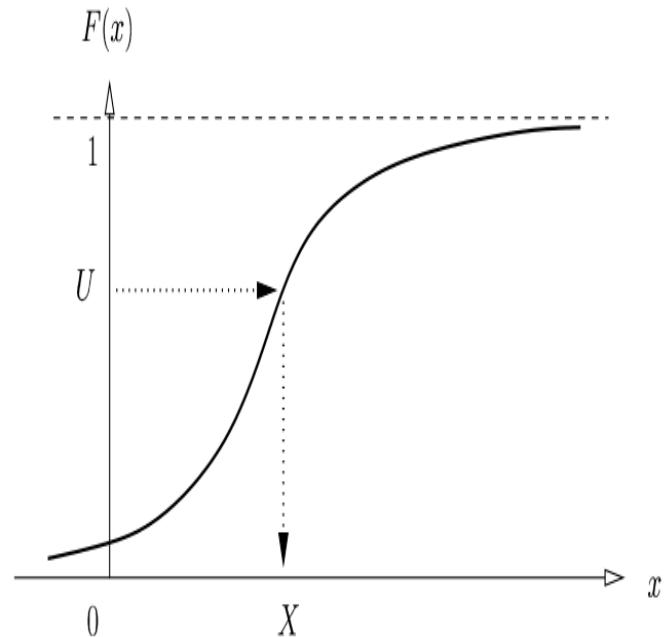
Demonstração

- $X = F^{-1}(U)$ tem por função de distribuição acumulada $F(x)$??
- Por definição $F(x) = P(X \leq x)$
- $P(X \leq x) = P(F^{-1}(U) \leq x)$
- $= P(U \leq F(x))$
- $= F(x)$ porque $P(U \leq a) = a$

Algoritmo

1. Gerar U com distribuição $U(0,1)$

2. Devolver $X = F^{-1}(U)$



Exemplo de aplicação – Simulação de uma variável aleatória **exponencial**

- Sendo $F(x) = 1 - e^{-x}$ (exponencial de média 1)
- $F^{-1}(u)$ será o valor de x que verifique
$$1 - e^{-x} = u$$
- ou seja $x = -\log(1 - u)$
- Portanto:

$$F^{-1}(u) = -\log(1 - u)$$

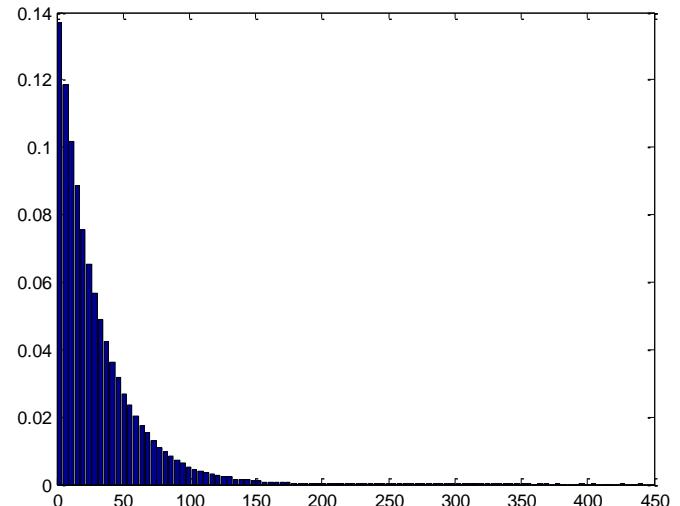
É exponencialmente distribuída com média 1

- $1-U$ é também uniforme em $(0,1)$
- Como $c X$ é exponencial com média c para obter uma exponencial de média c basta usar $-c \log(U)$

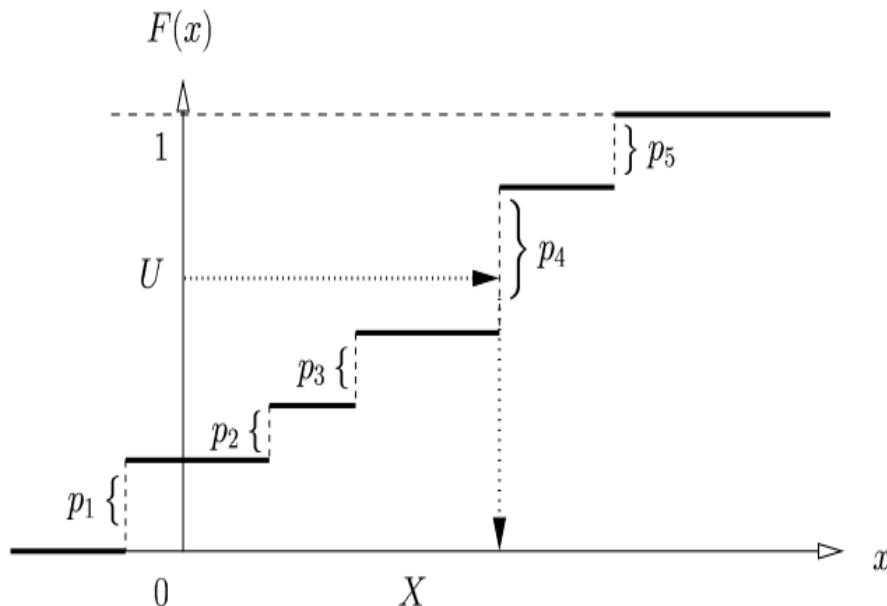
Exemplo em Matlab

```
function X=exponencial(m,N)
U=rand(1,N);
X=-m*log(U)

%
N=1e6
X=exponencial(10,N);
[n,xout] = hist(X,100);
bar(xout,n/N)
```



Algoritmo para caso discreto



1. Gerar U com distribuição $U(0,1)$ exemplo $U=0,7$
2. Ir aumentando x e determinar o primeiro para o qual $F(x) \geq U$
3. Devolver esse valor de x

A procura pode ser tornada mais rápida usando técnicas de procura eficientes

Método de procura numa tabela

- Se a função **cumulativa for guardada numa tabela**, então este algoritmo pode ser visto como uma simples procura numa tabela de

$$i \text{ tal que } F_{i-1} < u \leq F_i$$

- Ou seja:

$$X = \begin{cases} x_1, & \text{if } U < P_1 \\ x_2, & \text{if } P_1 < U < P_1 + P_2 \\ \vdots & \\ x_j, & \text{if } \sum_{i=1}^{j-1} P_i < U < \sum_{i=1}^j P_i \\ \vdots & \end{cases}$$

Exemplo de aplicação

- Gerar pseudo-palavras com as letras assumindo a probabilidade das letras em Português
 - Que já vimos anteriormente

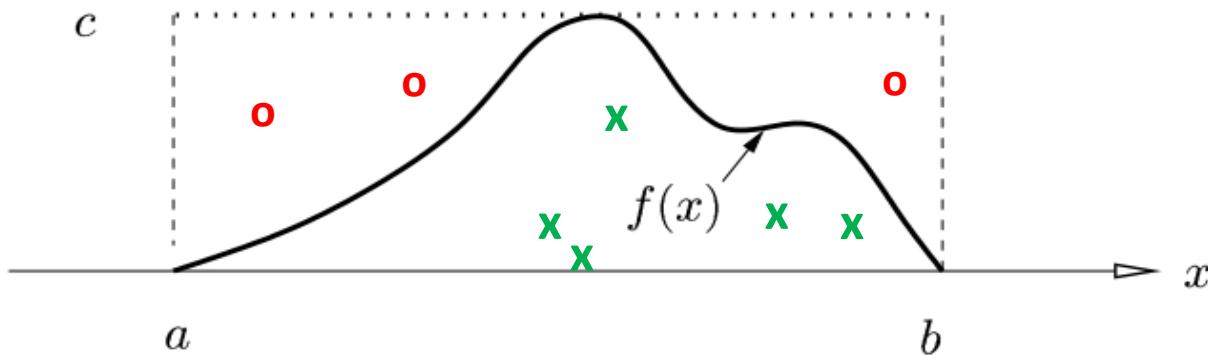
Em Matlab

```
letters='abcde';
% p=[0.0828  0.0084  0.0201  0.0342  0.0792]; % PT real
p=[0.800  0.01  0.01  0.01  0.17];           % fake
p=p/sum(p); % só existem para nós 5 letras

X= zeros(1,60);
for j=1:60
    U=rand();
    i = 1 + sum( U > cumsum(p) );
    % out sera valor entre 1 e 5
    % de acordo com as probabilidades p
    X(j)= letters(i);
end
char(X)
```

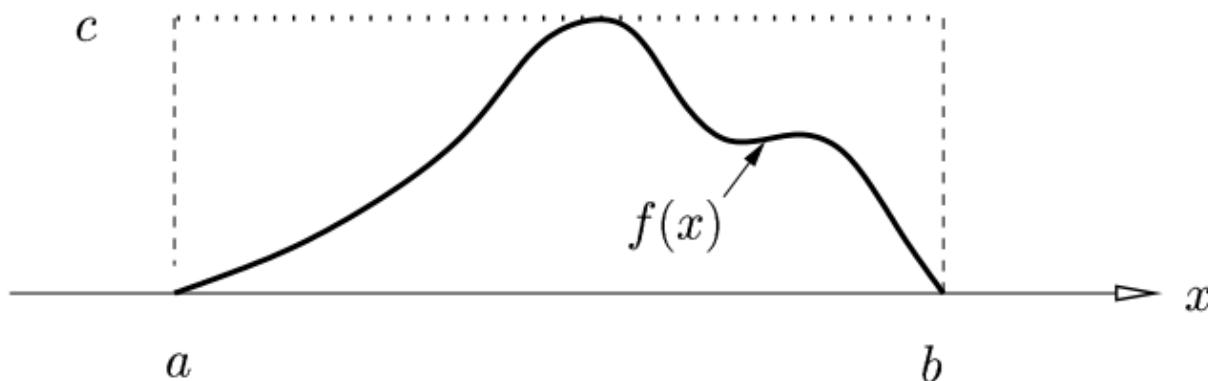
Métodos baseados em Rejeição

- Na sua forma mais simples:
 - define-se uma zona que contém todos os valores da função densidade de probabilidade no intervalo em que está definida
 - Geram-se números com distribuição uniforme nessa zona e rejeitam-se os que ficam acima de $f(x)$



Algoritmo

1. Gerar X com distribuição $U(a, b)$
2. Gerar Y com distribuição $U(0, c)$ independente de X
3. Se $Y \leq f(X)$ devolver $Z = X$; Caso contrário ir para o passo 1

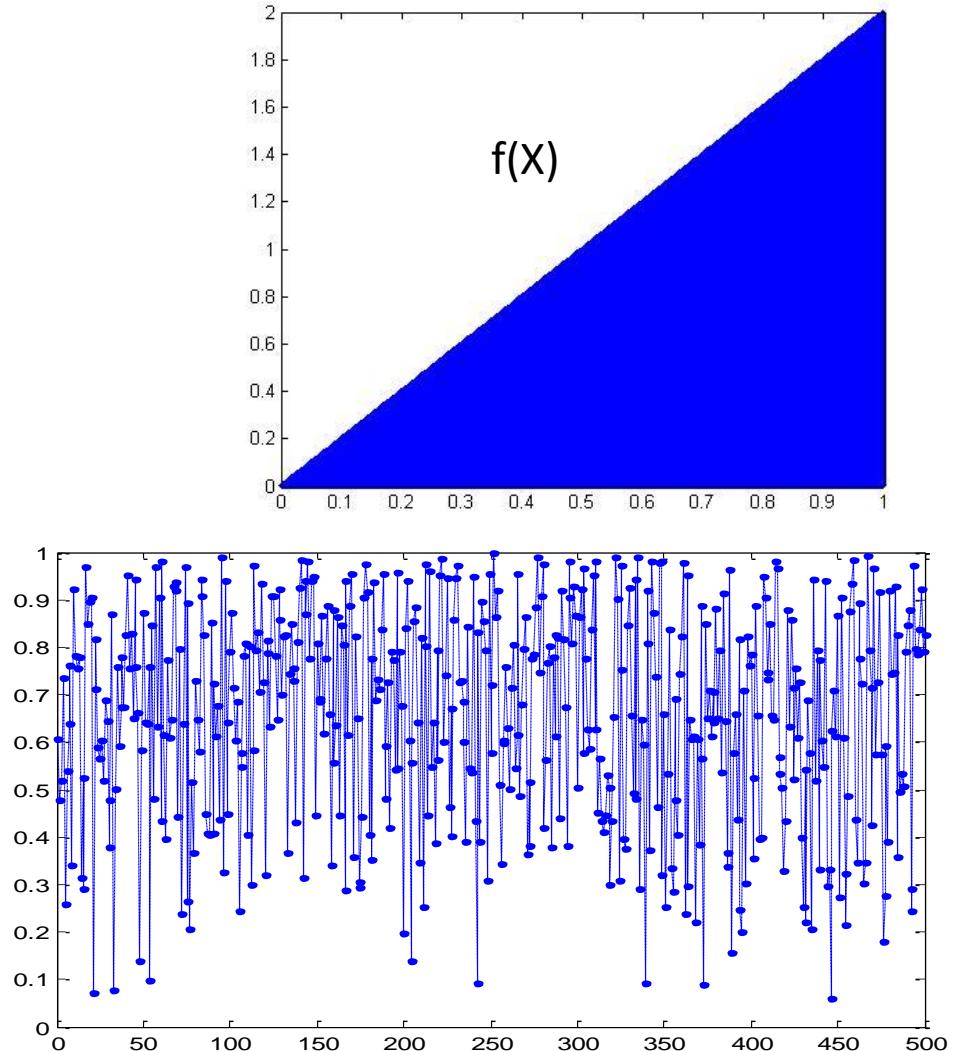


Exemplo

- $f(x) = \begin{cases} 2x & 0 \leq x \leq 1 \\ 0 & \text{outros valores} \end{cases}$
- Temos de usar $c=2$, $a=0$ e $b=1$

```
%  
N=1e6;  
X=rand(1,N);  
Y=rand(1,N)*2;
```

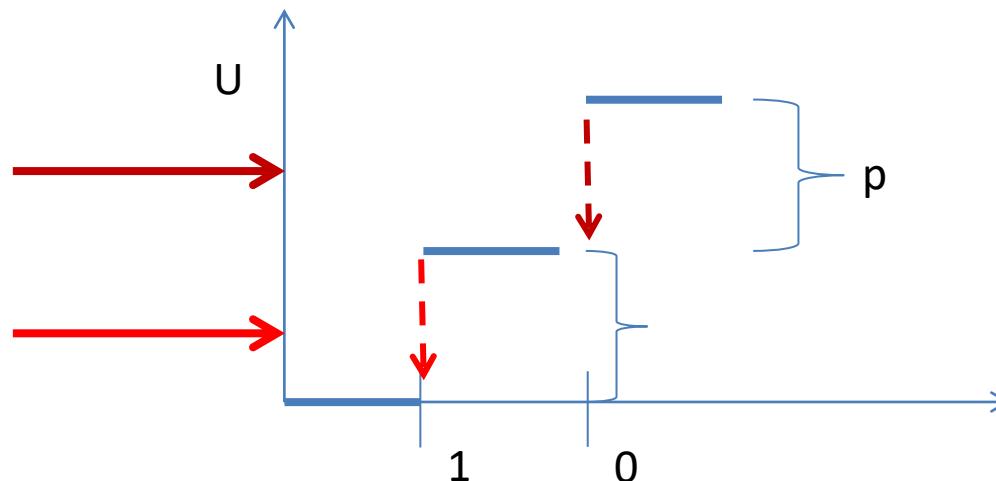
```
Z=X(Y<=2*X);  
  
% grafico  
Y2= Y(Y<=2*X);  
plot(Z,Y2,'.')
```



Algoritmos específicos para distribuição mais comuns (discretas)

Bernoulli

- Aplicando o método da transformação inversa para o caso discreto tem-se



- De onde decorre o seguinte algoritmo:
 - 1 – Gerar U com distribuição $U(0,1)$
 - 2 – Se $U \leq p$ $X=1$; caso contrário $X=0$

Exemplo Matlab

```
function X=Bernoulli (p,N)
```

```
X=rand(1,N)<=p
```

% usando

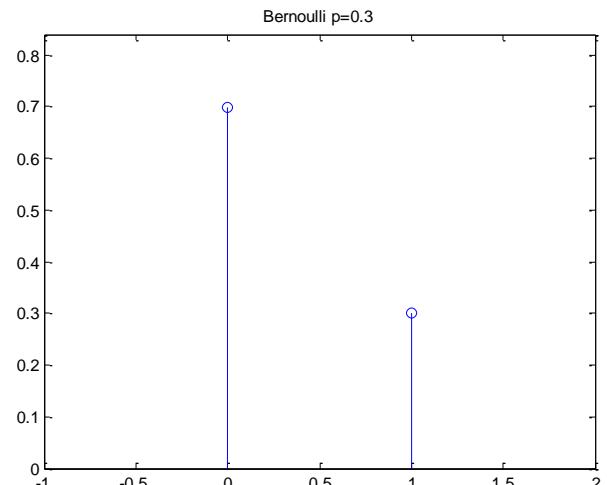
N=1e6

```
X=Bernoulli(0.3, N);
```

```
myhist(X,'Bernoulli p=0.3')
```

```
p=sum(X==1) /N
```

→ 0.2999



Técnicas especiais - Obter Binomial

- Pode obter-se uma variável aleatória Binomial usando o facto de que esta pode ser expressa como a **soma de n variáveis de Bernoulli independentes**
- $X = \sum_{i=1}^n X_i$ é uma v.a. Binomial com parâmetros n e p quando X_i é de Bernoulli com parâmetro p

Obter Binomial - Algoritmo

- Gerar variáveis independentes e identicamente distribuídas (iid) X_1, \dots, X_n usando distribuição de Bernoulli com parâmetro p
- Devolver $X = \sum_{i=1}^n X_i$

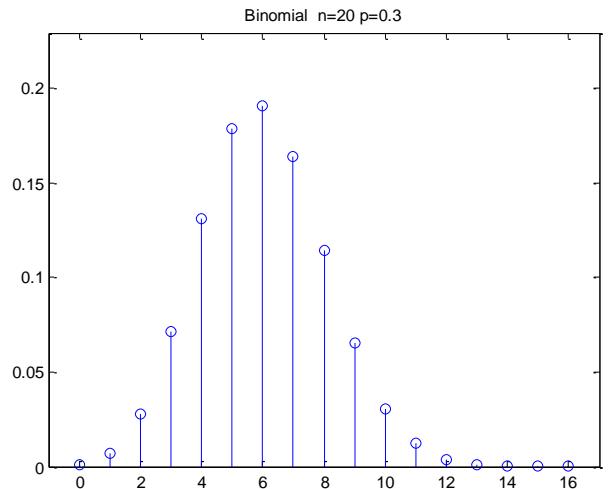
Demo obtenção binomial

```
function X=binomial(n,p, N)
Bern=rand(n,N)<=p; % n Bernoulli(p)
```

```
X=sum(Bern);
```

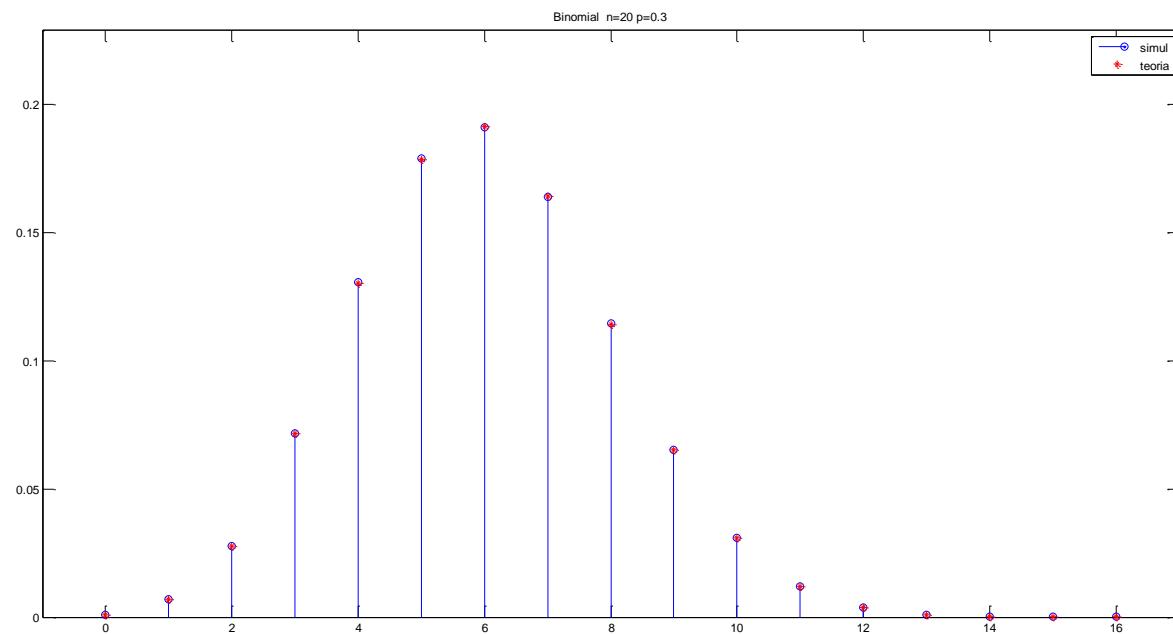
```
% usando
```

```
N=1e6; n=20; p=0.3;
X=binomial(n,p, N);
myhist(X,'Binomial n=20 p=0.3')
```



Simulação versus teoria

- $N=1e6$



Algoritmos específicos para distribuição mais comuns (contínuas)

Distrib. Normal – Alg. Box Müller

- Algoritmo de **Box e Müller**:

1 – Gerar 2 variáveis independentes U_1 e U_2 uniformes em $(0,1)$

2 – Obter 2 variáveis com distribuição Normal, X e Y , através de:

$$X = (-2 \ln U_1)^{1/2} \cos(2\pi U_2) ,$$

$$Y = (-2 \ln U_1)^{1/2} \sin(2\pi U_2) .$$

Box Müller em Matlab

```
function[X,Y]=BoxMuller(N)
```

```
U1=rand(1,N); % gerar uma v.a. uniforme
```

```
U2=rand(1,N); % gerar outra v.a. uniforme
```

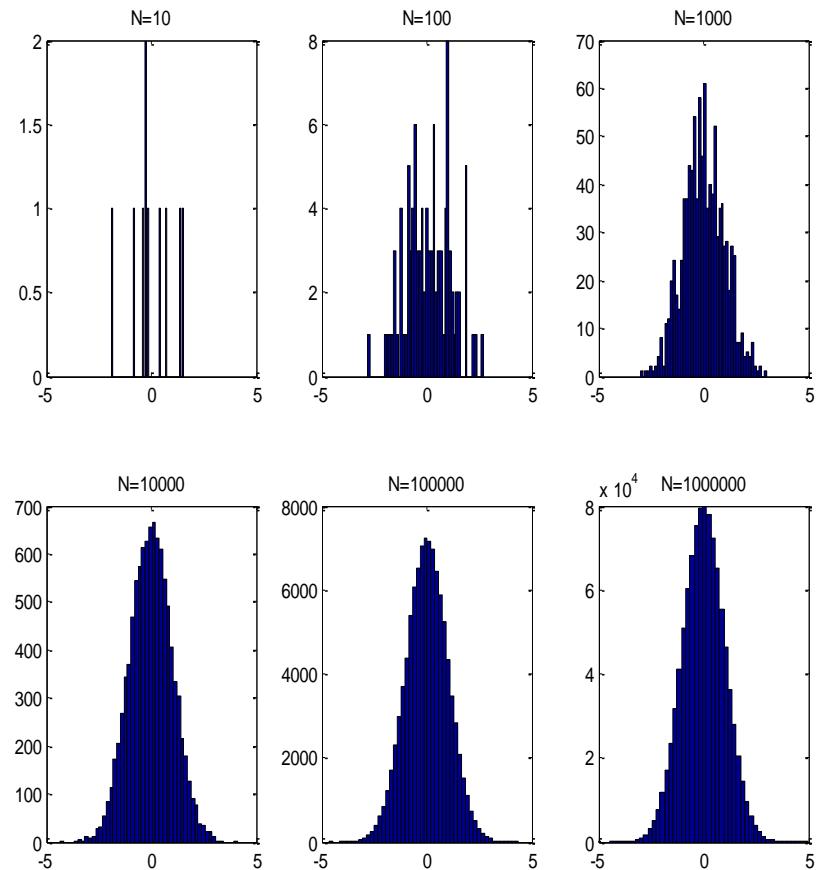
```
X=(-2*log(U1)).^(1/2).* cos(2*pi*U2);
```

```
Y=(-2*log(U1)).^(1/2).* sin(2*pi*U2);
```

- Atenção ao uso de $.^$ e $.*$

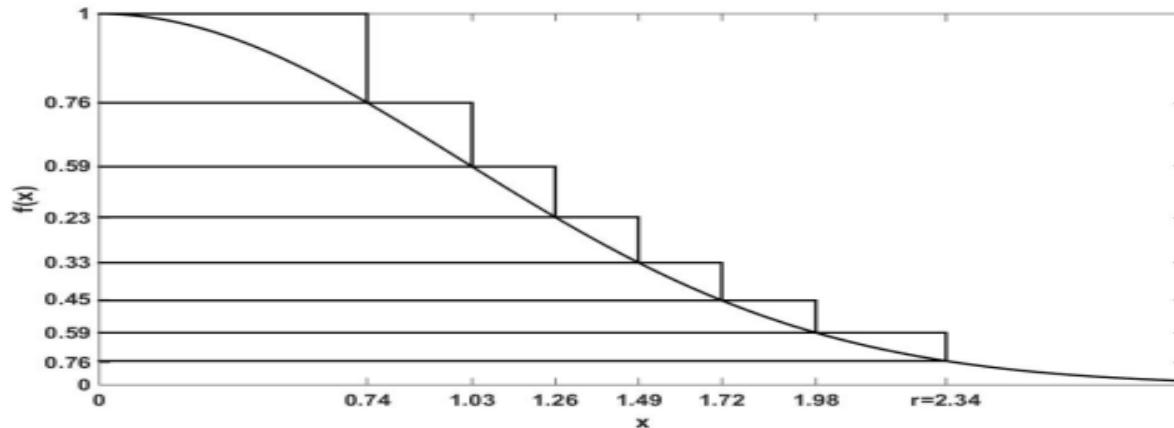
Demonstração em Matlab

```
for i=1:6
    subplot(2,3,i)
    N=10^i;
    [X,Y]=BoxMuller(N);
    hist(X,50)
    title(['N=' num2str(N)]);
    ax=axis;
    ax(1)=-5; ax(2)=5;
    axis(ax)
end
```



Distribuição Normal – Algoritmo Ziggurat

- Desenvolvido por Marsaglia em 2000
- É um método de rejeição
- Utiliza a curva $y = f(x) = e^{-x^2/2}$ para $x > 0$
 - Devido a simetria
- Utiliza um conjunto de tiras com a mesma área e geração de números com distrib. Uniforme



- Figura faz lembrar um Zigurate (antiga Mesopotâmia)

Em Java

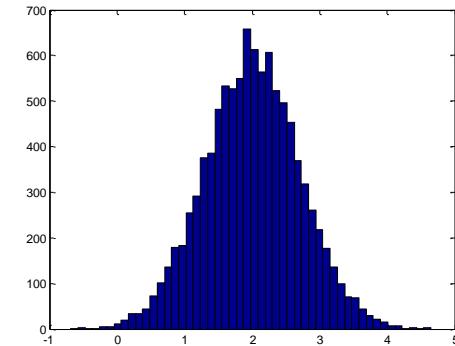
- É similar a gerar números de uma distribuição uniforme
- O exemplo seguinte mostra como gerar um número aleatório de uma distribuição Gaussiana com média 0 e variância 1

```
import java.util.*;  
Random r = new Random();  
g = r.nextGaussian();
```

- De cada vez que se invoca `r.nextGaussian()` obtém-se um novo número

Distribuição normal no Matlab

- Em Matlab está disponível a função **randn()**
 - Gera números aleatórios com uma distribuição Normal de média 0 e variância 1
- Para obter outras médias e variâncias basta aplicar uma transformação
- O comando randn() utiliza o algoritmo Ziggurat

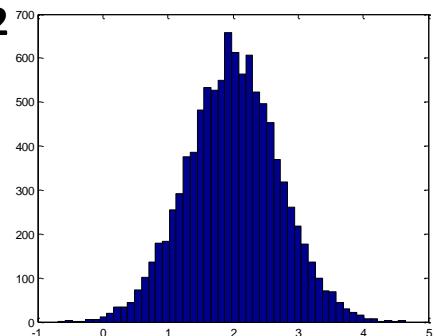


randn() Matlab

- Utilizando as já referidas propriedades
 $E(X + c) = E(X) + c$
e $Var(cX) = c^2 Var(X)$

podem gerar-se valores de distribuições com média e variância arbitrárias

- Exemplo: média 2 e variância $\frac{1}{2}$
`Y=sqrt(1/2) * randn(1, 1e4)+2;`
`hist(Y,50)`



Outras distribuições em Matlab

- Exemplos de distribuições **discretas**

Distribution	Random Number Generation Function
Binomial	binornd, random, randtool
Geometric	geornd, random, randtool
Negative binomial	nbinrnd, random, randtool
Poisson	poissrnd, random, randtool
Uniform (discrete)	unidrnd, random, randtool

- Fonte: <https://www.mathworks.com/help/stats/random-number-generation.html>

Outras distribuições em Matlab

- Exemplos de distribuições **contínuas**

Distribution	Random Number Generation Function
Chi-square	chi2rnd, random, randtool
Exponential	exprnd, random, randtool
Gamma	gamrnd, randg, random, randtool
Normal (Gaussian)	normrnd, randn, random, randtool
Rayleigh	raylrnd, random, randtool
Student's t	trnd, random, randtool
Uniform (continuous)	unifrnd, rand, random

- Fonte: <https://www.mathworks.com/help/stats/random-number-generation.html>

Para aprender mais

- Online
 - Capítulo “RANDOM NUMBERS, RANDOM VARIABLES AND STOCHASTIC PROCESS GENERATION”
[http://moodle.technion.ac.il/pluginfile.php/220739/mod_resource/content/0/slava fall 2010/Random number 2 .pdf](http://moodle.technion.ac.il/pluginfile.php/220739/mod_resource/content/0/slava_fall_2010/Random_number_2.pdf)
- Cap. 1 e Apêndice B do livro “Probabilidades e Processos Estocásticos”, F. Vaz, Universidade de Aveiro

MPEI

Funções de dispersão *(Hash functions)*

Motivação

- Em muitos programas de computador torna-se necessário aceder a informação através de uma chave
 - Exemplo:
 - Obter nome associado a um número de telefone
- Em Java, por exemplo, temos estruturas de dados como HashMap e Hashtable

Um dicionário simples: Hashtable

- Para criar uma *Hashtable*:

```
import java.util.*;  
Hashtable table = new Hashtable();
```

- Para colocar elementos (par chave-valor) na Hashtable, usa-se:

```
table.put(chave, valor);
```

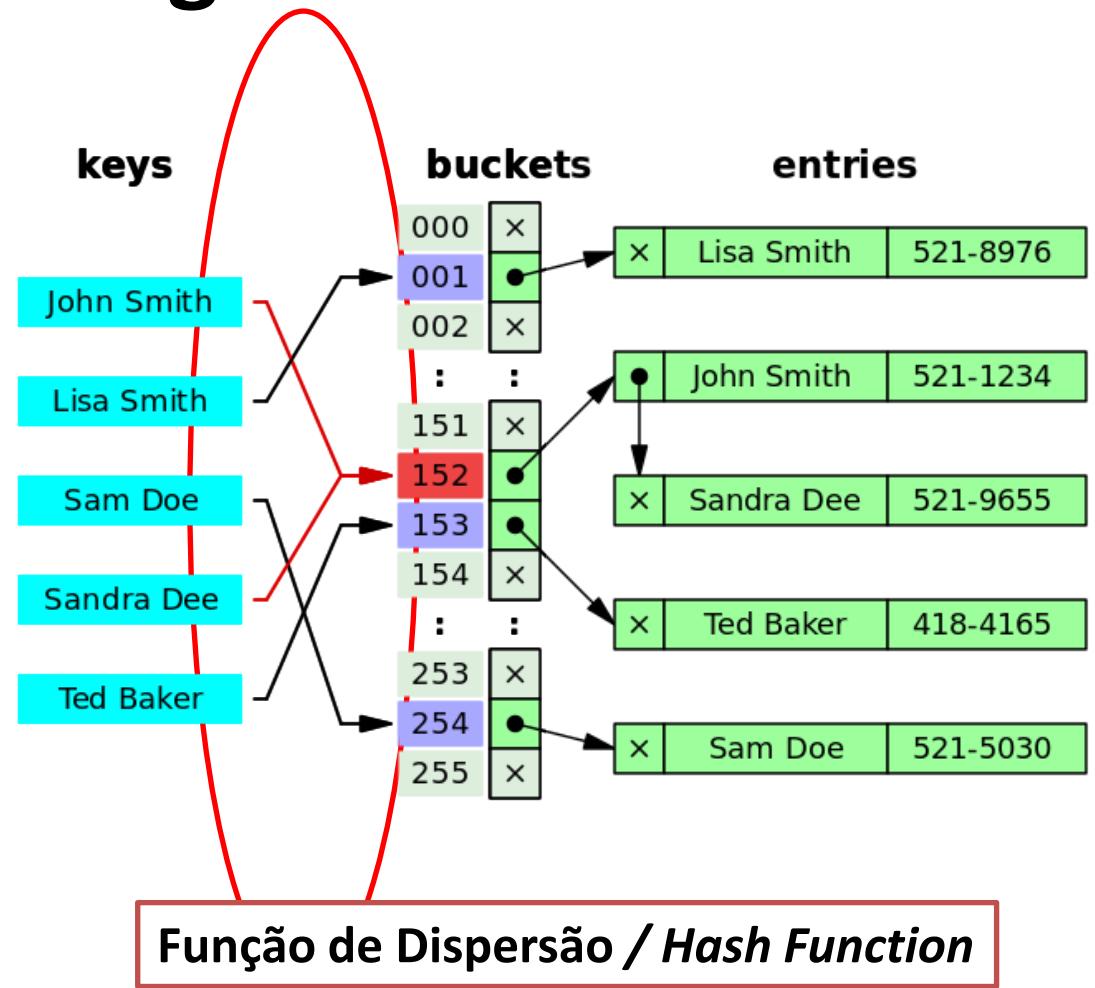
- Para obter um valor:

```
valor = table.get(chave);
```

Implementação comum

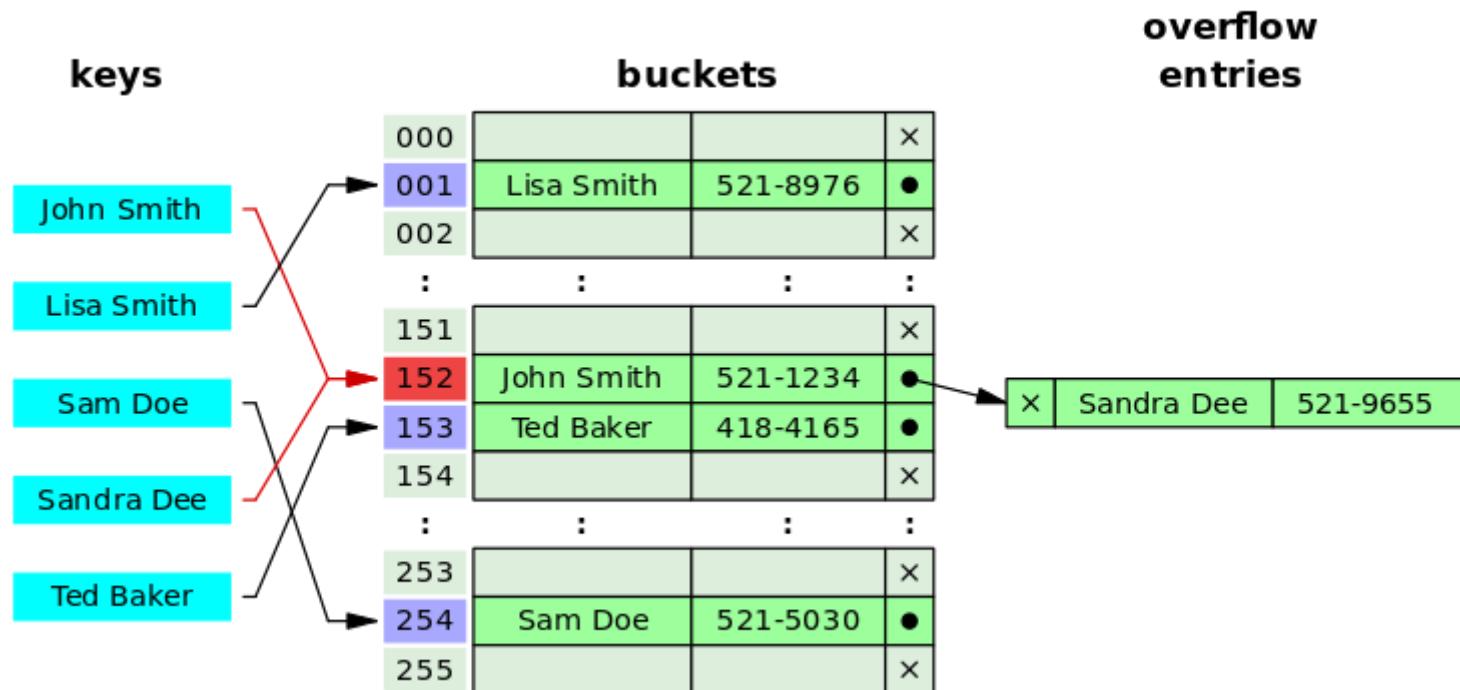
Separate chaining with linked lists

- As chaves são transformadas em posições num array
 - usando uma função
- Cada posição do array é o início de uma lista ligada



Outra implementação

Separate chaining with list head cells

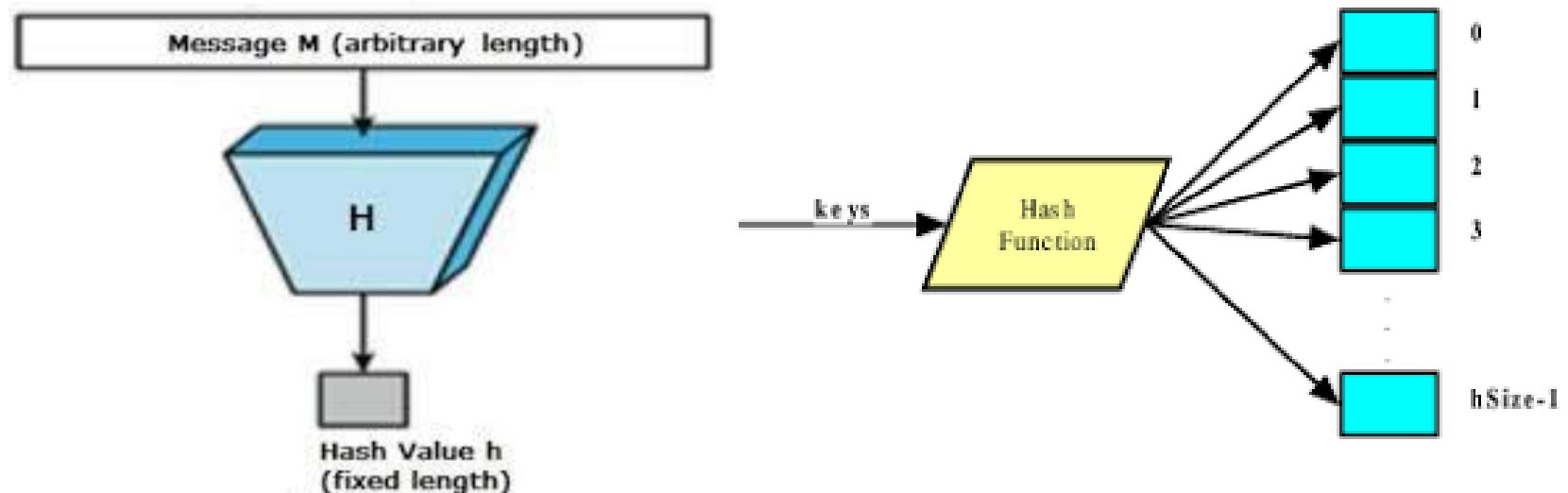


Função de dispersão

- Em termos gerais, uma função de dispersão - - em Inglês ***hash function*** - é qualquer algoritmo que **mapeia um conjunto grande e de tamanho variável para um conjunto de tamanho fixo de menor dimensão**
- É, como veremos, essencial para muitas aplicações

Função de dispersão / Hash function

- Uma função de dispersão (hash function) **mapeia** símbolos de um **universo U** num conjunto de M valores, em geral inteiros



- Processo pode ser visto como a atribuição de uma posição num vetor de M posições, entre 0 e M-1, a cada símbolo.
 - As **posições** designam-se muitas vezes por *buckets*

Hash Code

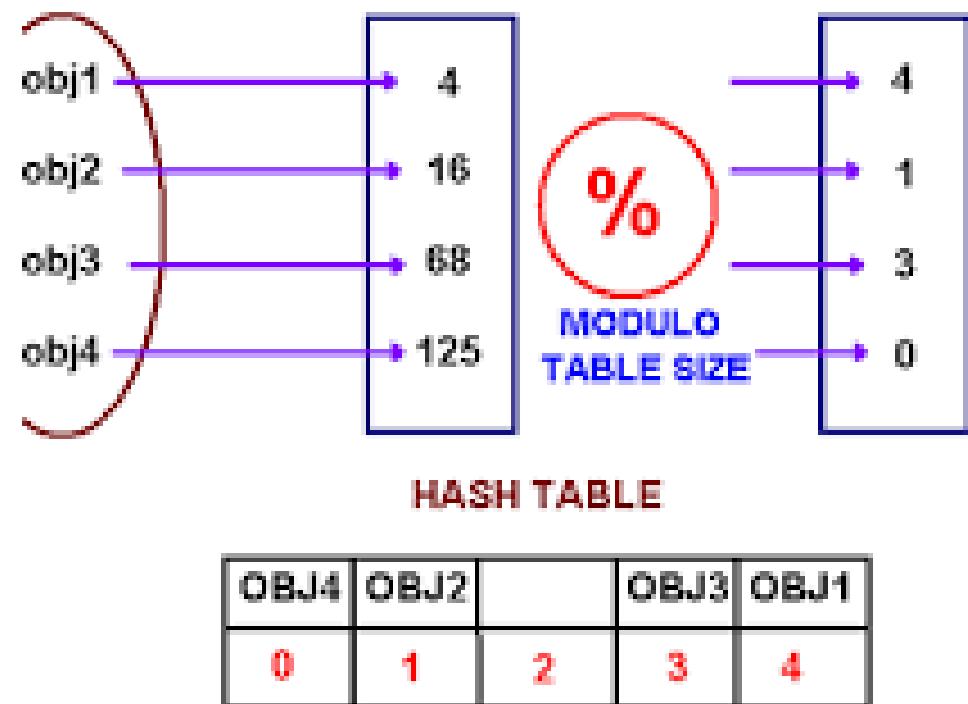
- O conjunto dos símbolos efetivamente usados numa determinada aplicação é, em geral, apenas uma parte do universo de valores (U) pelo que faz todo o sentido usar um **valor de M muito menor do que a dimensão de U**
 - Muitas vezes os valores designam-se por **chaves**
- Uma função de dispersão recebe um elemento de U como entrada e devolve um número inteiro h no intervalo $0, \dots, M - 1$
 - h é o Código de dispersão (em Inglês **hash code**)

Funções de dispersão / Hash functions

- Qualquer função que mapeie uma chave do universo U no intervalo $0..M - 1$ é uma função de dispersão em potencial.
- No entanto, uma tal função só é eficiente se **distribuir as chaves pelo intervalo de uma forma razoavelmente uniforme**
 - mesmo quando existem regularidades nas chaves.
- Uma função de dispersão ideal mapeia as chaves em inteiros de **uma forma “aleatória”**
 - De forma a que as *keys* sejam igualmente distribuídos pelos *buckets*.
- É fundamental que a função de dispersão seja uma função no sentido matemático do termo,
 - Isto é, que para cada chave a função devolva sempre o mesmo código

Funções de dispersão

- O processo pode ser dividido em dois passos:
1. Mapeamento do elemento para um inteiro
 2. Mapeamento do inteiro para um conjunto limitado (de inteiros).



Notação

- Adota-se para a representação das funções de dispersão $h()$
 - do Inglês hash function
- e k para uma chave
 - do Inglês key

Funções de dispersão - colisões

- Como o número de elementos de U é em geral maior que M , é inevitável que a função de dispersão mapeie **vários elementos diferentes no mesmo valor de h** , situação em que dizemos ter havido uma **colisão**
- Por exemplo, sendo k um elemento de U e a função de dispersão:
$$h(k, M) = k \bmod M$$
- teremos colisões para $k, M + k, 2M + k, \dots$

Exemplo muito simples

Considere o universo U é o conjunto dos números inteiros que vai de 100001 a 999999. Suponha que $M = 100$ e se adota os dois últimos dígitos da chave como código de dispersão (em outras palavras, o código é o resto da divisão por 100). Calcule os códigos (*hash codes*) para 123456, 7531 e 3677756.

Resultado:

chave	código
123456	56
7531	31
3677756	56

Propriedades

- Requer-se, em geral, que as funções de dispersão satisfaçam algumas propriedades, como:
- Serem determinísticas
- Uniformidade:
 - Uma boa função de dispersão deve mapear as entradas esperadas de forma igual por toda a gama de valores possíveis para a sua saída
 - Todos os valores possíveis para a função de dispersão devem ser gerados com aproximadamente a mesma probabilidade

Funções de dispersão para inteiros

- Estas funções mapeiam uma única chave inteira k num número inteiro $h(k)$ entre M possíveis
- Existem vários tipos:
 - baseadas em divisão
 - baseadas em multiplicação
 - membros de famílias universais.

Método da Divisão

- Utiliza o resto da divisão por M

- A função de dispersão é

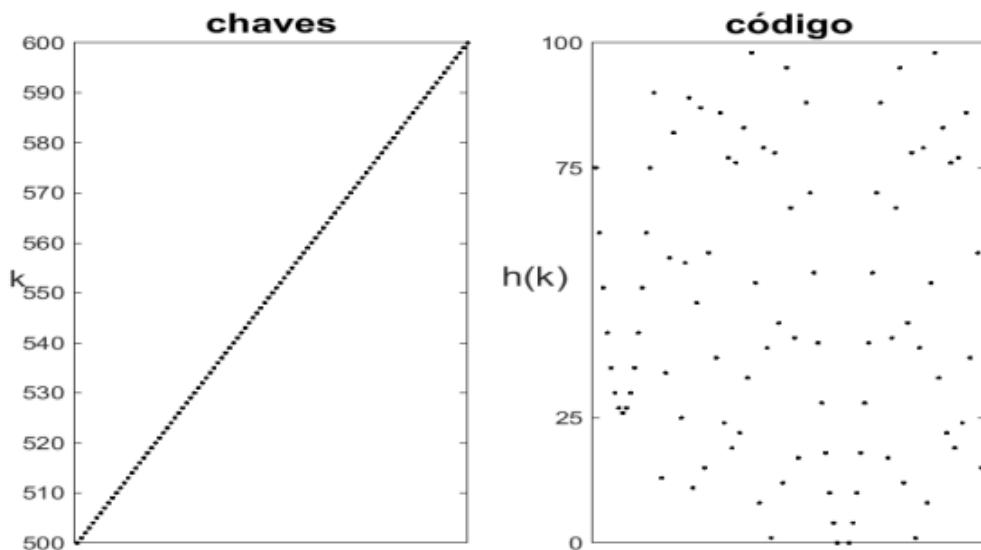
$$h(k) = k \bmod M$$

- M é o número de posições (igual ao tamanho da tabela), que deve ser um número primo
- Exemplo: se $M = 11$ e a chave $k = 100$ temos $h(k) = 1$
- Método bastante rápido
 - Requer apenas uma operação de divisão
- Funciona muito mal para muitos tipos de padrões nas chaves
- Foram desenvolvidas variantes como a de Knuth:

$$h(k) = k(k + 3) \bmod M$$

Exemplo: Variante de Knuth

- $h(k) = k(k + 3) \bmod M$
- $M = 113$
- Aplicação a todos os inteiros de 500 a 600.
- A sequência igualmente espaçada de números (à esquerda) é dispersada sem regularidade aparente
 - que é o que se pretende de uma boa função de dispersão



Método da multiplicação

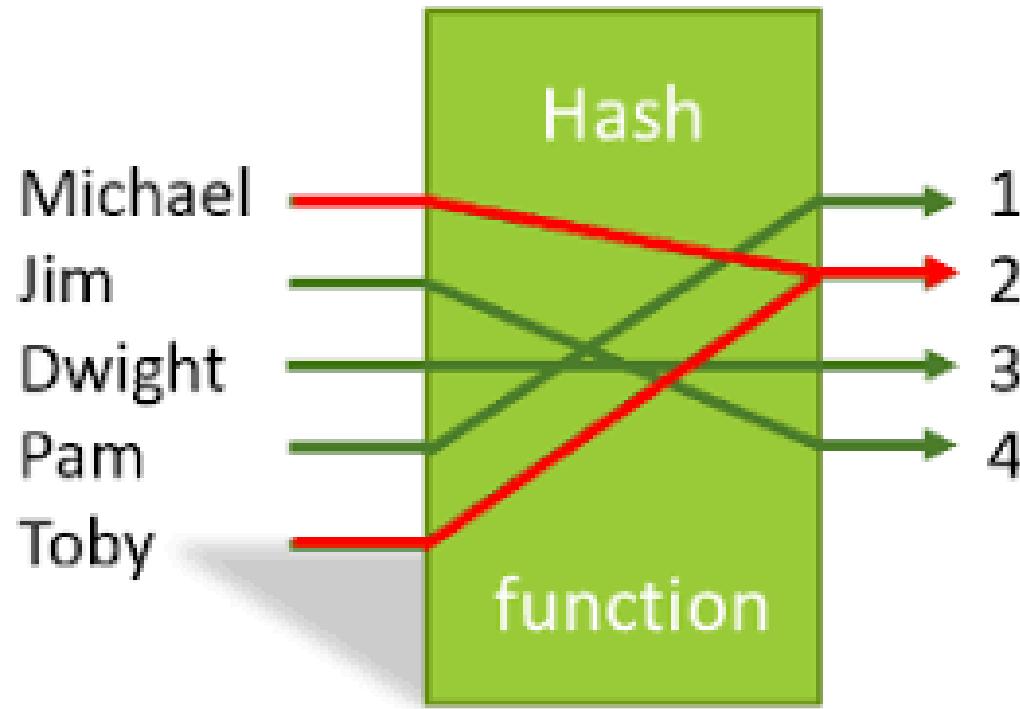
- Este método opera em duas etapas:
 - primeiro, multiplica-se a chave por uma constante A , $0 < A < 1$, e extrai-se a parte fraccionária de kA ;
 - de seguida,multiplica-se por M e arredonda-se para o maior inteiro menor ou igual ao valor obtido
- Matlab:

```
function h = hmultiplic(k,M)
% Função de dispersão para baseada na multiplicação
% Entradas: k - chave;
%             M - núm. de valores possíveis [0,M-1]
```

```
A= 0.5*(sqrt(5) - 1);
```

```
h=floor(M*(mod(k*A,1)));
```

Função de dispersão de uma sequência de caracteres



Função de dispersão de uma sequência de caracteres

- Uma função de dispersão para cadeias de caracteres (strings) calcula a partir desta, qualquer que seja o seu tamanho, um inteiro
- Como sabemos uma sequência de caracteres (String) é em geral representada como uma sequência de inteiros
- Em consequência, a função de dispersão para Strings tem por entrada uma sequência de inteiros
$$k = k_1, \dots, k_i, \dots, k_n$$
- e produz um número inteiro pequeno $h(k)$
- Os algoritmos para este tipo de entrada **assumem que os inteiros são de facto códigos de caracteres**

Função de dispersão de uma sequência de caracteres

- Os algoritmos para este tipo de entrada fazem em geral o uso do seguinte:
- Em muitas linguagens um caracter é representado em 8 bits
- O código ASCII apenas usa 7 desses 8 bits
- Desses 7, os caracteres comuns apenas usam os 6 menos significativos
 - E o mais significativo desses 6 indica essencialmente se é maiúscula ou minúscula, muitas vezes pouco relevante
- Em consequência os algoritmos **concentram-se na preservação do máximo de informação dos 5 bits menos significativos**, fazendo muito menos uso dos 3 bits mais significativos

Função de dispersão de uma sequência de caracteres (String)

- Em geral, o processamento efetuado consiste em:
 - inicializar h com 0 ou outro valor inicial
 - Percorrer a sequência de inteiros (representando os caracteres) combinando os inteiros ki , um por um, com h
 - Os algoritmos diferem na forma como combinam ki com h
 - Obtenção do resultado final através de $h \bmod M$ (método da divisão).
- Para evitar ao máximo problemas com overflow, em geral os inteiros ki são representados por números inteiros sem sinal (unsigned int)
 - A utilização de representações de inteiros com sinal pode resultar em comportamentos estranhos

Exemplo simples

$$\text{hash}(\text{key}) = \sum_{i=0}^{\text{KeySize}-1} \text{Key}[\text{KeySize}-i-1] \cdot 37^i$$

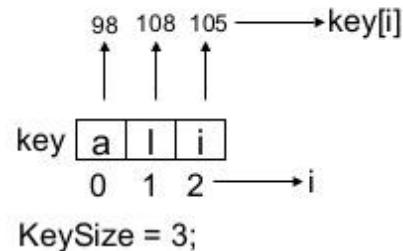
```
int hash (const string &key, int tableSize)
{
    int hashVal = 0;

    for (int i = 0; i < key.length(); i++)
        hashVal = 37 * hashVal + key[i];

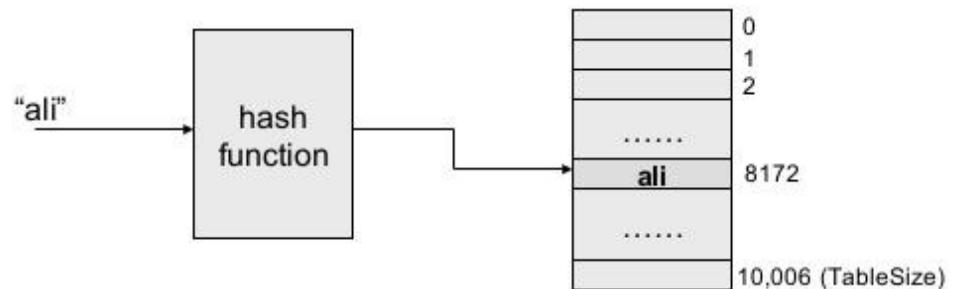
    hashVal %= tableSize;
    if (hashVal < 0) /* in case overflows occurs */
        hashVal += tableSize;

    return hashVal;
};
```

Hash function for strings:



$$\text{hash}(\text{"ali"}) = (105 * 1 + 108 * 37 + 98 * 37^2) \% 10,007 = 8172$$



Exemplo – hashCode() do Java

- A classe `java.lang.String` implementa desde o Java 1.2 a função `hashCode()` usando um somatório de produtos envolvendo todos os caracteres
- Uma instância `s` da classe `java.lang.String` tem o seu código `h(s)` definido por:

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

- com `s[i]` representando o código UTF-16 do carácter i da cadeia de comprimento n
- A adição é efectuada usando 32 bits

Alguns algoritmos – variante CRC

- Fazer um shift circular de 5 bits para a esquerda ao h.
- De seguida fazer XOR de h com ki.

CRC variant: Do a 5-bit left circular shift of h. Then XOR in ki. Specifically:

```
highorder = h & 0xf8000000      // extract high-order 5 bits from h
                                // 0xf8000000 is the hexadecimal representation
                                // for the 32-bit number with the first five
                                // bits = 1 and the other bits = 0
h = h << 5                      // shift h left by 5 bits
h = h ^ (highorder >> 27)        // move the highorder 5 bits to the low-order
                                // end and XOR into h
h = h ^ ki                        // XOR h and ki
```

Alguns Algoritmos– PJW hash

- Deslocar (shift) h 4 bits para a esquerda
- Adicionar ki
- Mover 4 bits mais significativos

PJW hash (Aho, Sethi, and Ullman pp. 434-438): Left shift h by 4 bits. Add in ki. Move the top 4 bits of h to the bottom. Specifically:

```
// The top 4 bits of h are all zero
h = (h << 4) + ki           // shift h 4 bits left, add in ki
g = h & 0xf0000000          // get the top 4 bits of h
if (g != 0)                  // if the top 4 bits aren't zero,
    h = h ^ (g >> 24)       //   move them to the low end of h
    h = h ^ g
// The top 4 bits of h are again all zero
```

Exemplo de uma função completa

- Mapeia uma string de comprimento arbitrário num inteiro (≥ 0)
- DJB31MA

```
uint hash(const uchar* s, int len, uint seed)
{
    uint h = seed;
    for (int i=0; i < len; ++i)
        h = 31 * h + s[i];
    return h;
}
```

- Fonte: Paulo Jorge Ferreira “MPEI – summary” 2014

Exemplo Matlab

function hash=string2hash(str,type)

```
% This function generates a hash value from a text string
%
% hash=string2hash(str,type);
%
% inputs,
% str : The text string, or array with text strings.
% outputs,
% hash : The hash value, integer value between 0 and 2^32-1
% type : Type of has 'djb2' (default) or 'sdbm'
%
% From c-code on : http://www.cse.yorku.ca/~oz/hash.html
.....
```

- From: <http://www.mathworks.com/matlabcentral/fileexchange/27940-string2hash/content/string2hash.m>

Exemplo Matlab

```
str=double(str);  
  
hash = 5381*ones(size(str,1),1);  
  
for i=1:size(str,2),  
    hash = mod(hash * 33 + str(:,i), 2^32-1);  
end
```

Exemplos de uso ($M = 11$):

$k = \text{António}$	$\rightarrow h(k) = 4$
$k = \text{Antónia}$	$\rightarrow h(k) = 1$
$k = \text{Manuel}$	$\rightarrow h(k) = 6$
$k = \text{Manu}$	$\rightarrow h(k) = 4$
$k = \text{Manuela}$	$\rightarrow h(k) = 0$
$k = \text{Vitor}$	$\rightarrow h(k) = 0$

Problemas

- As funções de dispersão terão que lidar com conjuntos $S \subseteq U$ com $|S| = n$ chaves não conhecidos de antemão
- Normalmente, o objetivo destas funções é obter um número baixo de colisões (chaves de S que mapeiam na mesma posição)
- Uma função de dispersão determinística (fixa) não pode oferecer qualquer garantia de que não ocorrerá o pior caso:
 - um conjunto S com todos os elementos a serem mapeados na mesma posição, tornando a função de dispersão inútil em muitas situações.
- Além disso, uma função determinística não pode ser alterada facilmente em situações em que ocorram muitas colisões.

Solução

- A solução para estes problemas consiste em **escolher uma função aleatoriamente de uma família de funções,**
- Têm particular interesse as famílias de funções universais.

Funções de dispersão universais

- Uma família H de funções de dispersão h é universal se:

$$\forall x, y \in U, x \neq y : P_{h \in H}[h(x) = h(y)] \leq \frac{1}{M}$$

- Por palavras...
- **quaisquer duas chaves do universo colidem com probabilidade máxima igual a $1/M$** quando a função de dispersão h é extraída aleatoriamente de H
 - exatamente a probabilidade de colisão esperada caso a função de dispersão gerasse códigos realmente aleatórios para cada chave.

Funções de dispersão universais

- Esta solução garante um baixo número de colisões em média, mesmo no caso de os dados serem escolhidos por alguém interessado na ocorrência do pior cenário (ex: *hacker*).
- Este tipo de funções pode utilizar mais operações do que as funções que vimos anteriormente
- Existe uma diversidade de famílias universais e métodos para as construir
 - Veremos a seguir alguns

Método de Carter Wegman

- A proposta original, de Carter e Wegman, consiste em escolher um primo $p \geq M$ e definir

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod M$$

- sendo a e b inteiros aleatórios módulo p ($a \neq 0$)
- Trata-se de uma iteração de um gerador de números aleatórios de congruência linear.

Método da Matriz

- Este método baseia-se em:
 1. considerar as **chaves na sua representação binária**
 2. construir uma matriz de bits aleatoriamente
 3. multiplicar a chave e matriz

Método da Matriz (continuação)

- Consideremos que as chaves são representáveis por u bits
- Sendo M uma potência de 2 : $M = 2^b$
- Criar uma matriz h de 0s e 1s de forma aleatória
 - a matriz terá dimensões $b \times u$
- Definir $h(x) = hx$
 - usando adição mod 2
- Exemplo:
 - $u = 4$
 - $b = 3$

$$\begin{array}{c} h \quad \quad x \quad \quad h(x) \\ \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{array} \right] \quad \left[\begin{array}{c} 1 \\ 0 \\ 1 \end{array} \right] = \left[\begin{array}{c} 1 \\ 1 \\ 0 \end{array} \right] \end{array}$$

O que significa hx ?

- Pode ser interpretada como a adição de algumas das colunas de h , aquelas em que x tem o valor 1 nas linhas

$$\begin{array}{c|cccc} & h & & x & h(x) \\ \hline & 1 & 0 & 0 & 0 \\ & 0 & 1 & 1 & 1 \\ & 1 & 1 & 1 & 0 \\ \hline & & & 1 & 1 \\ & & & 0 & 0 \end{array}$$

- No exemplo:
- A 1^a e 3^a coluna são somadas
 - $1 + 0 = 1$
 - $0 + 1 = 1$
 - $1 + 1 = 0$

Método da matriz

Propriedade. A função de dispersão $h(x)$ definida desta forma terá:

$$\forall x \neq y, \quad P_{h \in H}[h(x) = h(y)] = \frac{1}{M} = \frac{1}{2^b}$$

Demonstração:

- Um par de chaves diferentes x e y difere em algum dos bits.
Consideremos que diferem no bit na posição i e que $x_i = 0$ e $y_i = 1$.
- Se selecionarmos toda a matriz h exceto a coluna i obteremos um valor fixo para $h(x)$;
- No entanto, cada uma das 2^b diferentes possibilidades da coluna i implica um valor diferente para $h(y)$, pois sempre que se muda um valor nessa coluna muda o bit correspondente em $h(y)$;
- Em consequência temos exatamente a probabilidade $1/2^b$ de $h(x) = h(y)$.

Outro método

- Mais eficiente do que o da matriz
- A chave é representada por um **vetor de inteiros**

- Em vez do vetor de bits do método da matriz

$$[x_1, x_2, \dots, x_k]$$

- x_i pertencendo a $\{0, 1, \dots, M - 1\}$
 - k é o tamanho do vetor
 - **M um número primo**

- Exemplo:
 - Em Strings, x_i pode representar o código do caractere i

Outro método (continuação)

- Para seleccionar uma função de dispersão h escolhem-se k números aleatórios

r_1, r_2, \dots, r_k de $\{0,1, \dots, M - 1\}$

- E define-se :

$$h(x) = (r_1x_1 + r_2x_2 + \dots + r_k x_k) \bmod M$$

Exemplo Matlab

```
s='Métodos Probabilísticos'  
M= 113;  
  
% converter para vetor  
x=double(s)  
  
% gerar vetor r  
r=randi(M-1,1,length(x))  
  
%  $h(x) = r * x \bmod M$   
h=mod( r*x', M)
```

Demonstração da universalidade

- A demonstração segue a mesma linha da apresentada anteriormente para o método da matriz
- Considere-se duas chaves distintas x e y
- Pretendemos demonstrar que

$$P[h(x) = h(y)] \leq 1/M$$

Demonstração da universalidade

Como $x \neq y$, existe pelo menos um índice i tal que $x_i \neq y_i$. Selecionando todos os números aleatórios r_j com $j \neq i$ podemos definir $h'(x) = \sum_{j \neq i} r_j x_j$.

Desta forma, ao escolher um valor para r_i teremos $h(x) = h'(x) + r_i x_i$.

Teremos uma colisão entre x e y exatamente quando

$$h'(x) + r_i x_i = h'(y) + r_i y_i \text{ mod } M$$

ou, de forma equivalente, quando

$$r_i(x_i - y_i) = h'(y) - h'(x) \text{ mod } M .$$

Demonstração da universalidade

Como M é primo, a divisão por um valor não nulo módulo M é possível e existe apenas um único valor $r_i \ mod \ M$ que constitui a solução, mais exactamente

$$r_i = \frac{h'(y) - h'(x)}{x_i - y_i} \ mod \ M$$

.....

Temos assim apenas uma possibilidade de igualdade entre 1 e $M - 1$ e, em consequência, a probabilidade de colisão é exatamente $1/M$, como pretendíamos demonstrar.



Exemplo em Matlab

```
%  
function InitHashFunction(this)  
    % Set prime parameter  
    ff = 1000; % fudge factor  
    pp = ff * max(this.m + 1, 76);  
    pp = pp + ~mod(pp, 2); % make odd  
    while (isprime(pp) == false)  
        pp = pp + 2;  
    end  
    this.p = pp; % sufficiently large prime number  
  
    % Randomized parameters  
    this.a = randi([1, (pp - 1)]);  
    this.b = randi([0, (pp - 1)]);  
    this.c = randi([1, (pp - 1)]);  
end
```

Exemplo em Matlab - HashCode()

```
function hk = HashCode(this,key)
    % Convert character array to integer array
    ll = length(key);
    if (ischar(key) == false)
        % Non-character key
        HashTable.KeySyntaxError();
    end
    key = double(key) - 47; % key(i) = [1,...,75]

    %
    % Compute hash of integer vector
    %

    % Reference: http://en.wikipedia.org/wiki/Universal\_hashing
    %             Sections: Hashing integers
    %                         Hashing strings
    %

    hk = key(1);
    for i = 2:ll
        % Could be implemented more efficiently in practice via bit
        % shifts (see reference)
        hk = mod(this.c * hk + key(i),this.p);
    end
    hk = mod(mod(this.a * hk + this.b,this.p),this.m) + 1;
end
end
```

Como ter n funções de dispersão ?

Possíveis soluções:

1. Ter mesmo n funções diferentes
2. Usar funções customizáveis (definindo uma **família de funções**) e usando parâmetros diferentes
3. Usar a mesma função de dispersão e **processar a chave por forma a ter n chaves diferentes** baseadas na chave original

Exemplo (Matlab):

```
for i=1:n  
    str= [str num2str(i)];  
    h=HashCode(hash,m,str);  
end
```

Propriedades (continuação)

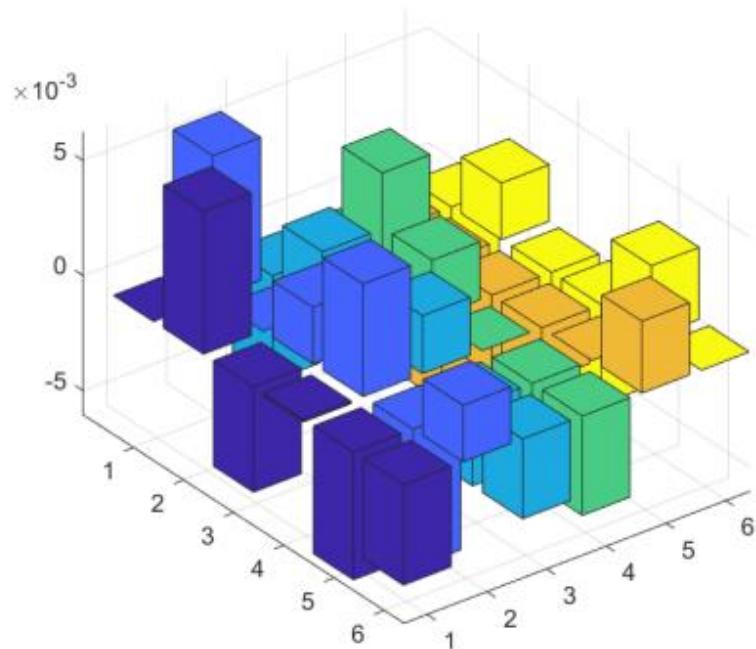
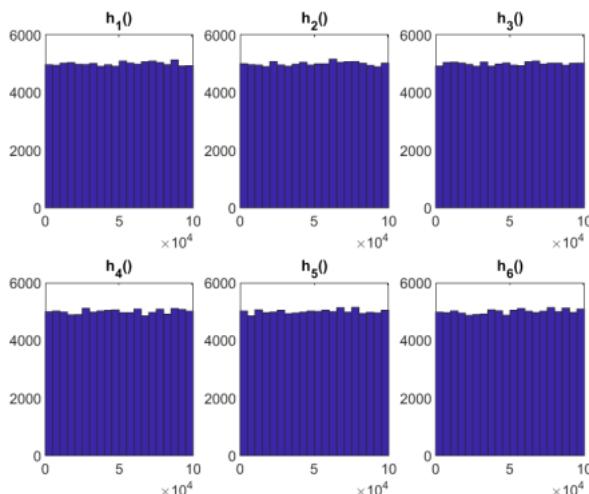
- As *n* funções de dispersão devem cumprir um requisito adicional:
- Produzir resultados não-correlacionados
- Esta propriedade é muito importante e é aconselhável verificá-la/avaliá-la em trabalhos envolvendo várias funções

“Teste” de funções de dispersão

- Um teste simples e básico consiste em:
 1. Gerar um conjunto grande de chaves (pseudo)aleatórias
 2. Processar todas essas chaves com as n funções de dispersão
 - Guardando os resultados produzidos (*hash codes*)
 3. Analisar o histograma de cada função de dispersão
 - Para verificar a uniformidade da distribuição dos *hash codes*
 4. Calcular, visualizar e analisar as correlações entre os resultados produzidos pelas várias funções de dispersão

Exemplo

- Teste com 100 mil números de 6 funções (h_1, \dots, h_6)



Funções de dispersão criptográficas

- Para este tipo de funções (cryptographic hash functions) M é um **número exponencialmente grande**
 - Como 2^{256}
- A tabela seria maior que o número de eletrões no universo!
- Mas não temos a tabela... $h(k)$ é apenas uma impressão digital (fingerprint) de k .
- Mesmo para M com valores muito grandes os índices de $h(k)$ são pequenos
 - Ex: apenas 256 bits (32 bytes) para $M = 2^{256}$
- **A principal propriedade** requerida para uma função de dispersão criptográfica é de que seja **computacionalmente intratável** para alguém **descobrir** $y \neq x$ tal que $h(y) = h(x)$

Alguns links

- <http://www.mathworks.com/matlabcentral/fileexchange/27940-string2hash/content/string2hash.m>
- <http://www.mathworks.com/matlabcentral/fileexchange/45123-data-structures/content/Data%20Structures/Hash%20Tables/HashTable.m>
- <http://www.cse.yorku.ca/~oz/hash.html>
- <http://programmers.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed>
- <https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html>
- https://en.wikipedia.org/wiki/Hash_function#Properties
- https://en.wikipedia.org/wiki/Universal_hashing
- <http://www.i-programmer.info/programming/theory/2664-universal-hashing.html>

Funções de Dispersão Universais

- <https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1004.pdf>
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-8-universal-hashing-perfect-hashing/lec8.pdf>
- http://cs-www.bu.edu/faculty/homer/537/talks/SarahAdelBargal_UniversalHashingnotes.pdf

MPEI

Solução Probabilística
do
Problema da Pertença a um Conjunto

Definição do problema

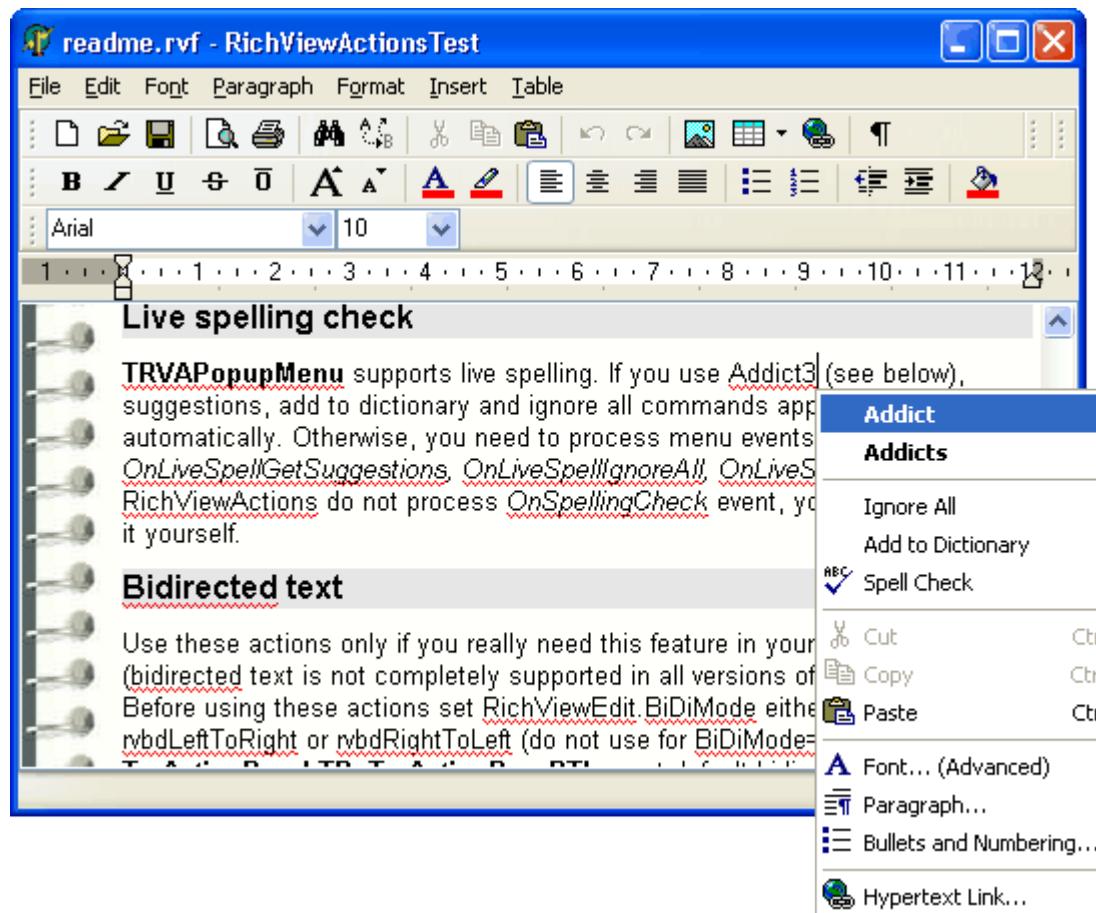
- Em termos gerais o problema pode ser colocado da seguinte forma:
- Dado um **elemento e** e um **conjunto C** , **e pertence a C ?**
- O elemento pode ser, por exemplo, uma String

Problema para conjuntos de grandes dimensões

- A resolução deste tipo de problemas para pequenos conjuntos é fácil
 - usando, por exemplo, *hash tables*
- No entanto, para **conjuntos de dimensão muito grande** (e apenas passíveis de serem definidos em extensão) não é assim tão simples
 - pode mesmo não haver memória suficiente para armazenar todos os elementos de C
 - aparecendo **soluções probabilísticas como necessárias e interessantes**
 - Com o aumento de Big Data cada vez mais relevantes

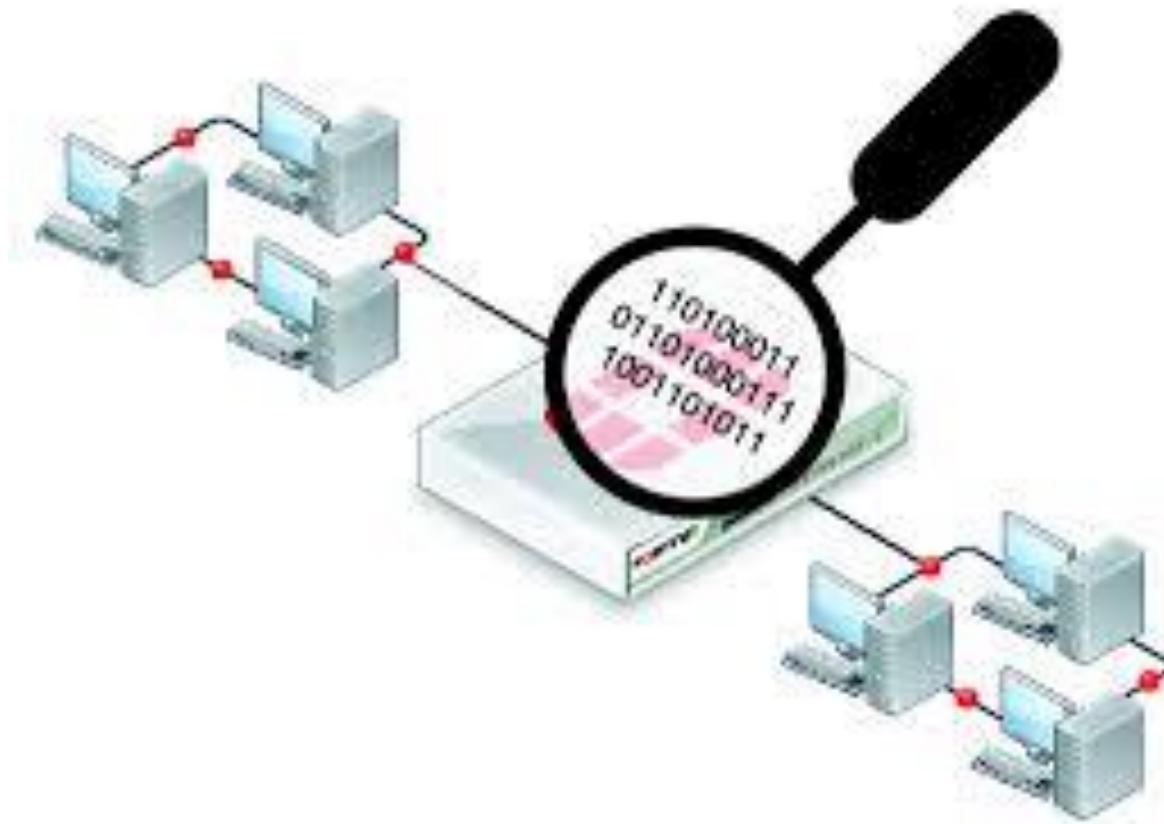
Problema 1 - Verificação ortográfica

- Utiliza dicionário(s)
 - Muitas vezes complementados por regras
- Ortografia validada através da pertença a dicionário
 - Que deve ser de confiança



Problema 3

- Detetar sequências de bits (ex: Strings)



Generalizando ...

- Na área da Informática temos muitos outros exemplos
 - Combate ao **SPAM**: pertença a uma lista de emails seguros
 - **Browser**: pertença à lista de endereços que já visitámos
 - ...
- Em muitas aplicações e situações temos de ter uma **forma eficiente** de **determinar se um determinado item pertence ou não a um conjunto**

Ideia base

- Em muitos problemas **apenas pretendemos saber se um elemento pertence ou não ao conjunto**
 - sem necessitarmos de acesso ao conjunto ou informação associada aos elementos
- Nestas situações **podemos eliminar a parte de armazenamento dos elementos**
 - guardando nele apenas informação de que existe

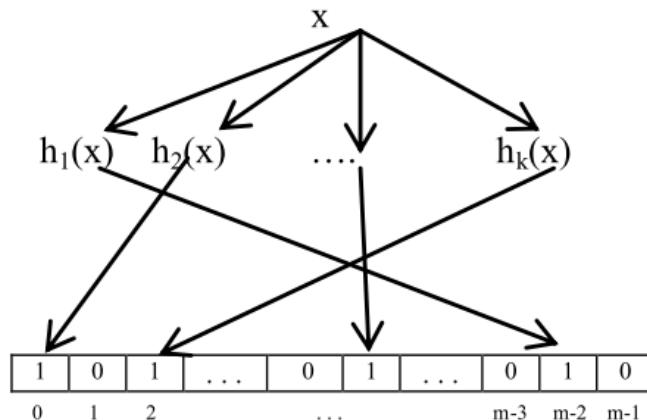
Filtros de Bloom

(Bloom Filters)

Os Filtros de Bloom são uma forma de usar funções de dispersão para determinar se um elemento pertence a um conjunto

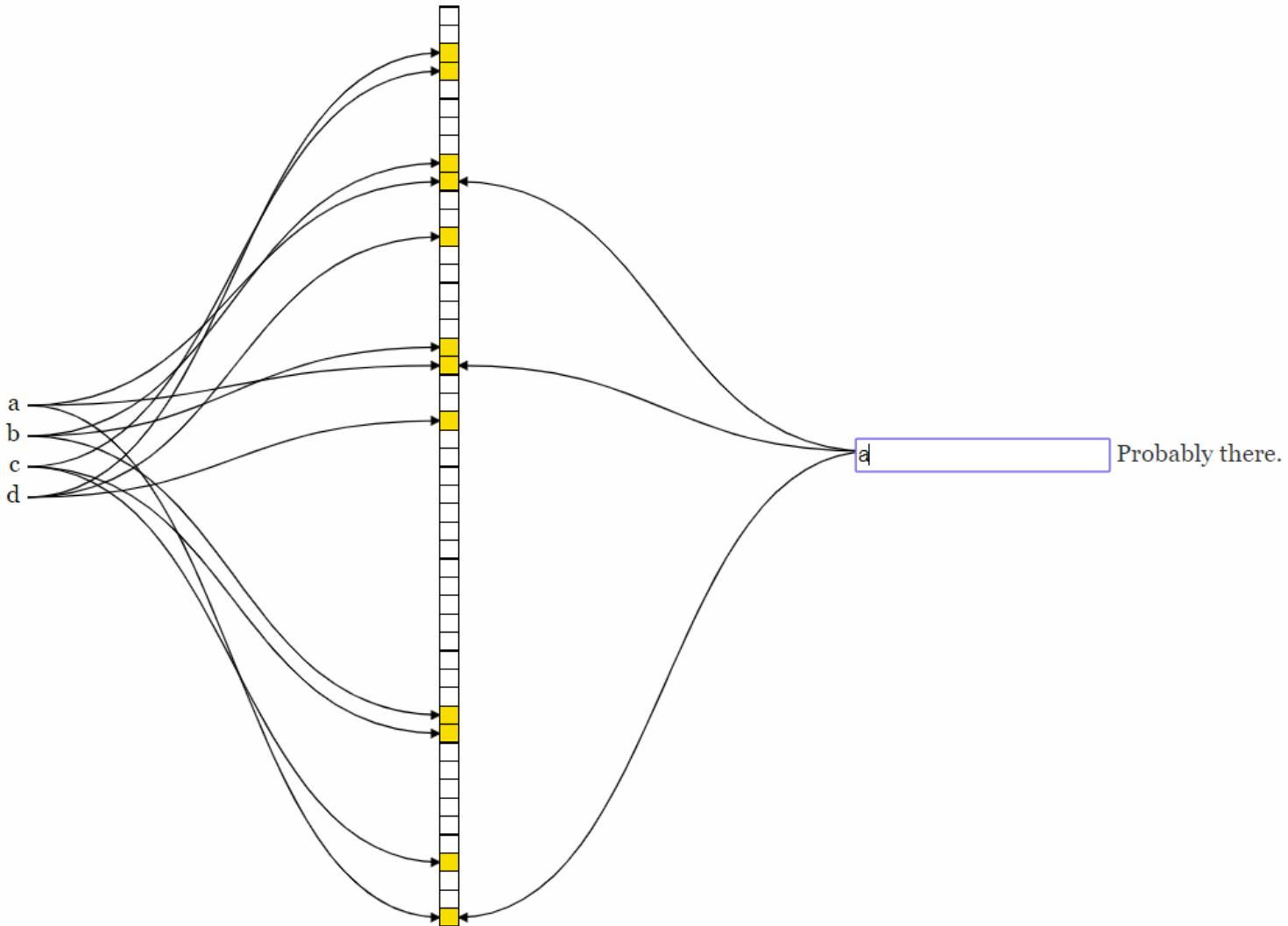
Filtros de Bloom

- Os Filtros de Bloom **usam funções de dispersão para calcular um vetor (o filtro)** que é representativo do conjunto



- A **pertença ao conjunto** é testada através da comparação dos resultados da aplicação das mesmas funções de dispersão aos potenciais membros com o conteúdo desse vetor

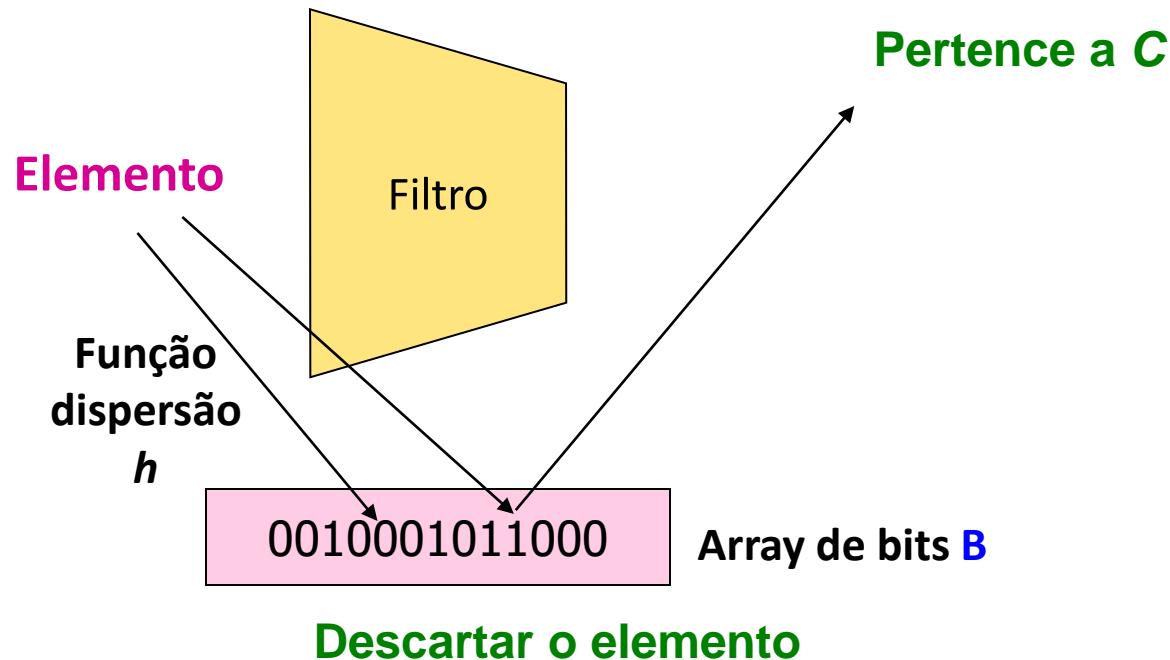
Filtro de Bloom



Filtros de Bloom

- Na sua **forma mais simples** o vetor (filtro) é composto por n posições
 - cada uma de apenas 1 bit
- O bit correspondente a um elemento é apenas colocado a 1 se a função de dispersão mapear nessa posição algum dos elementos do conjunto
- São rápidos, de complexidade temporal constante
- e não incorporam qualquer tentativa de resolução de colisões.

Filtro



Se é mapeado pela função de dispersão para uma posição contendo 0

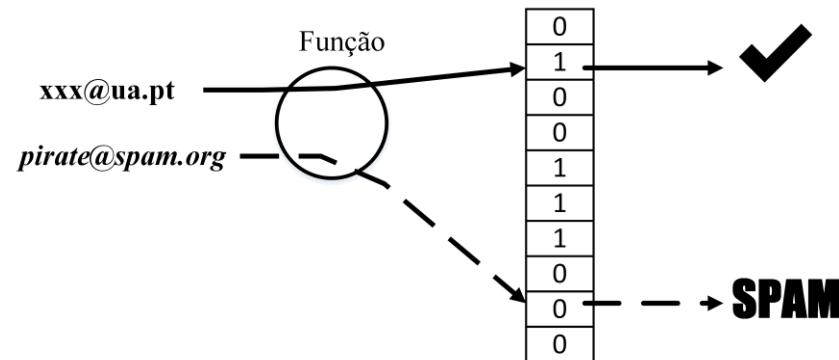
Exemplo – Deteção de SPAM

- Conhecem-se mil milhões (10^9) **de endereços de email de confiança**
 - que constituem o nosso conjunto C
- Se uma mensagem é proveniente de um destes endereços não é SPAM
- Pretende-se filtrar um conjunto de emails recebidos

Exemplo – Deteção de SPAM

- Uma solução consiste em:

1. Criar um vetor de n bits B bastante grande e inicializando todos esses bits com 0
2. Utilizar uma função de dispersão para mapear cada endereço de email numa posição desse vetor
3. Colocar a 1 os bits correspondentes à aplicação da referida função de dispersão a toda a lista de endereços de email “bons” (todos os elementos do conjunto C)
4. Aplicar a função de dispersão ao endereço de cada uma das mensagens que se pretende verificar, considerando SPAM todas as em que o vetor tem 0 na posição correspondente



Ausência de falsos negativos

- Se um endereço que verificamos pertence a C então ele vai ser certamente mapeado pela função de dispersão numa posição do vetor que contém 1
 - Será sempre considerado de confiança
- Todos os que são de confiança serão sempre considerados de confiança, **nunca havendo falsos negativos.**

Generalização

- A solução adotada para o exemplo anterior pode ser **generalizada pela utilização de um conjunto de funções de dispersão**
 - com algumas vantagens, como veremos mais adiante.
- No caso geral temos
 - C como sendo o **Conjunto**
 - com m elementos (membros) $(\#C = m)$
 - B o vetor (filtro) de **Bloom**
 - de dimensão n $(\#B = n)$
 - k funções de dispersão independentes h_1, \dots, h_k

Inicialização de um Filtro de Bloom

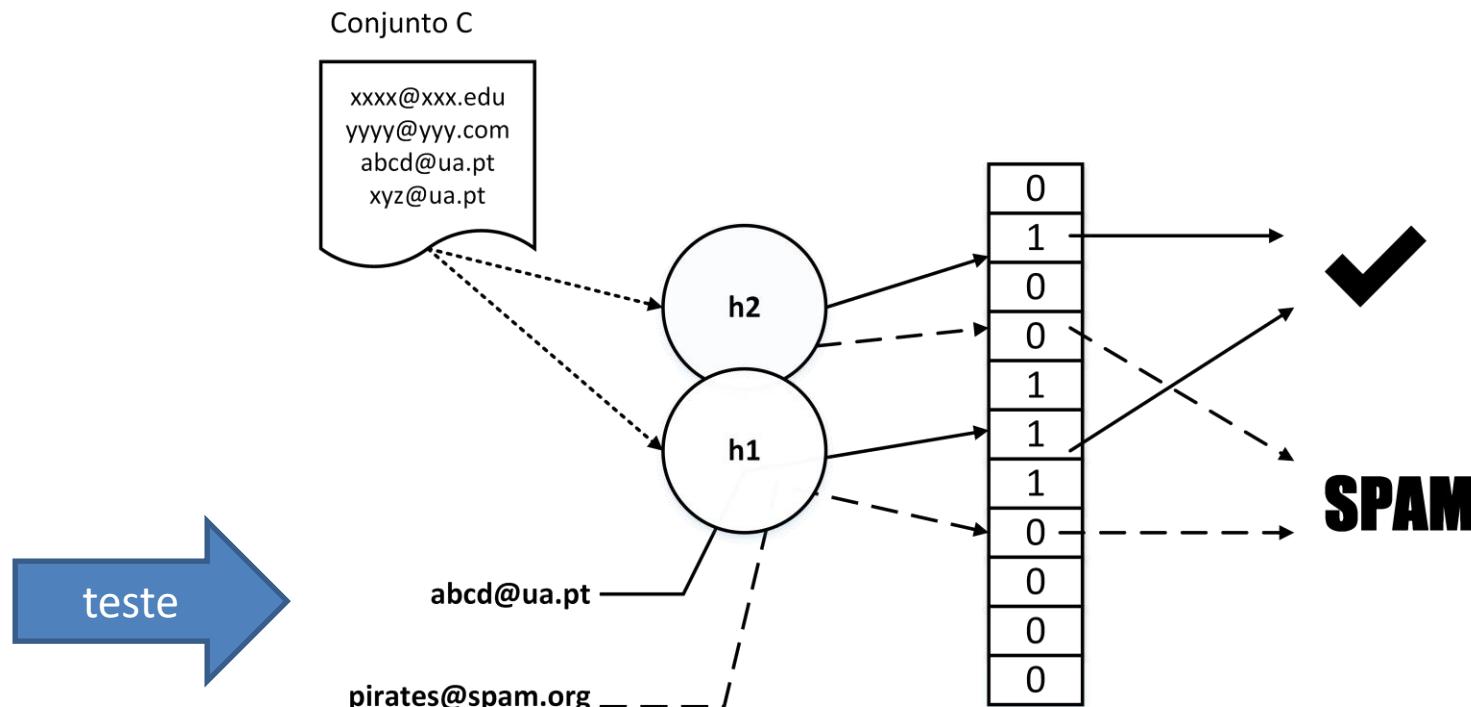
- Inicializar todas as posições de B com 0
- Aplicar k funções de dispersão a cada um dos elementos de C
 - colocando a um todas as posições devolvidas pelas funções de dispersão
 - Ou seja $B[h_i(elemento)] = 1.$

Utilização do Filtro de Bloom

- Para testar um valor x :
- aplicar as k funções de dispersão
- e analisar o conteúdo das posições resultantes
- se $B[h_i(x)] == 1$ para todos os valores de $i = 1, \dots, k$ então x provavelmente pertence a C
 - caso contrário não pertence C

Retomando o nosso exemplo ...

- Adotando valores pequenos para o número de funções, tamanho do conjunto e do filtro
 - $n = 11$ bits
 - $k = 2$ funções de dispersão.



Erros

- O teste de associação para um elemento x funciona verificando os elementos que teriam sido atualizados se a chave tivesse sido inserida no vetor
- Se todos os **bits apropriados foram colocados a 1 por outras chaves**, então x será reportado erradamente como um membro do conjunto.
- Temos neste caso o que se designa habitualmente por **falso positivo**.

Exemplo de falso positivo

- Consideraremos um conjunto de vegetais contendo batata e couve mas não tomate

Uma parte do filtro:

...	B	C	B	B	C	-	C	..
-----	---	---	---	---	---	---	---	----

Resultado de aplicação das funções de dispersão a Tomate:

...	-	T	-	T	T	-	-	..
-----	---	---	---	---	---	---	---	----

- O filtro da figura identifica incorretamente tomate como vegetal

Parâmetros do Filtro de Bloom

- m : Dimensão do conjunto
 - número de membros do conjunto
- n : Número de posições ou células do filtro
- k : Número de funções de dispersão utilizadas
- Adicionalmente, pode definir-se a fração das posições do filtro com o valor igual a 1 (f)

Implementação

- Em geral, a implementação destes filtros consiste em 3 operações
 - inicialização
 - adição de elementos
 - E teste de pertença de um elemento ao conjunto
- A implementação destas operações é simples

Tipo de dados abstrato

FiltroBloom

n % número de bits do filtro

m % número de elementos do conjunto

k % número de funções de dispersão

+ inicializar (n)

 % Inicializar filtro com 0s

+ adicionarElemento (elemento)

 % Inserir elemento no filtro

+ membro (elemento)

 % Testa se elemento existe no filtro

Implementação

- **inicializar()**
 - Simplesmente o preenchimento com zeros de todo o vetor B
- **adicionarElemento()**
 - Esta operação calcula os valores das k funções de dispersão do elemento a adicionar e atualiza as posições apropriadas do vetor B
 - No caso mais simples, coloca a 1 as posições devolvidas pelas funções de dispersão
 - o que requer tempo proporcional ao número de funções

adicionarElemento()

adicionarElemento(B, elemento,k)

for i=1:k do

- Aplicar a função de dispersão $hi()$ a elemento
 - Obtendo o respetivo hash code h
- Colocar a 1 o Vetor B na posição h
 - $B[h] = 1$

endfor

membro()

- Aplica as k funções como adicionarElemento(), mas **apenas verifica se as posições contêm o valor 1**
 - Se alguma das posições contém 0 não é um membro do conjunto
- A pior situação em termos de tempo de processamento ocorre para membros e para falsos positivos
 - Ambos obrigam a calcular todas as k funções de dispersão

membro()

membro(B, elemento) -> Boolean

i=0

repeat

i=i+1

hi é a função de dispersão i ($1 < i \leq k$)

h é o hash code devolvido por hi (elemento)

until ((i==k) | (B [h] == 0))

if i==k **then**

 return (B [h] == 1) % True se posição contém 1

else

 return(False)

end.

Complexidade das operações

Operação	Parâmetros	Complexidade temporal
Inicializar()	n (tamanho do vetor)	$O(n)$
adicionarElemento()	Vetor, elemento, k funções de dispersão	$O(k)$
membro()	Vetor, elemento, k funções de dispersão	$O(k)$

Exemplos de aplicações de Filtros de Bloom

Mais Informação em:

BLOOM FILTERS & THEIR APPLICATIONS, International Journal of Computer Applications and Technology (2278 - 8298) Volume 1– Issue 1, 2012, 25-29

<https://pdfs.semanticscholar.org/d899/05bdf1ff791bdddc7c471070f34f4da18844.pdf>

Aplicações gerais – *Spell Checkers*

- Os filtros Bloom são particularmente úteis na **verificação de ortografia**
 - São usados para determinar se uma palavra é válida numa determinada língua
- Verificação é feita **criando um Filtro Bloom com todas as palavras possíveis dessa linguagem** e verificando uma palavra contra esse filtro
- As correções sugeridas são geradas fazendo todas as substituições únicas em palavras rejeitadas e, em seguida, verificando se esses resultados são membros do conjunto

Redes de dados

- Muitas aplicações na área de redes
- São usados, por exemplo, em **caches de servidores proxy** na World Wide Web (WWW)
 - para **determinar eficientemente a existência de um objeto em cache**
 - Servidores proxy interceptam solicitações de clientes e respondem caso tenham a informação solicitada
- O uso de caches na web ajuda a **reduzir o tráfego da rede**
- Também **melhora o desempenho** quando os clientes obtêm cópias de arquivos de servidores vizinhos em vez do servidor original
 - que pode ser vários links de rede lentos de distância)

Segurança e Privacidade

- **Sistemas de detecção e prevenção de intrusão** (IDS/IPS) usam matching de strings do conteúdo dos pacotes para deteção de conteúdo malicioso
- Os filtros Bloom são particularmente úteis para pesquisar um grande número de strings de forma eficiente.
- A ideia básica é encontrar (sub)strings (comumente conhecidas como assinaturas) a alta velocidade
- Uma abordagem comum é separar assinaturas por comprimento e usar filtro Bloom para cada comprimento
 - permitindo processamento paralelo
- **O Google Chrome usa filtros Bloom** para a decisão preliminar se um determinado **site é malicioso ou seguro**
- Os filtros de Bloom também são usados na **deteção de vírus** e Prevenção de Negação de Serviço (DoS), entre muitas outras aplicações

Demonstrações Online

- **Bloom Filters by Example**
 - <http://billmill.org/bloomfilter-tutorial/>
- Bloom Filters
 - <https://www.jasondavies.com/bloomfilter/>

Exemplos de Aplicação (em Matlab)

- Pequeno exemplo com vegetais
- Dicionário
- Strings aleatórias
 - Efeito do número de funções de dispersão (k)
- Alunos da UC MPEI

Questões?

- Como obter n ?
- Como determinar o melhor valor para k (número de funções de dispersão) ?

Obtenção dos parâmetros de um Filtro de Bloom

Para que se possa definir de forma adequada um filtro tem de primeiro se perceber **como se relacionam com os indicadores de desempenho como a probabilidade de falsos positivos**

Falsos positivos e falsos negativos

Elemento pertence ao conjunto ?	Resultado do teste de pertença	Correto ?	Tipo de erro
SIM	Sim (pertence)	Sim	
	Não	ERRO	Falso negativo
NÃO	Sim	ERRO	Falso positivo
	Não	Sim	

Obtenção dos parâmetros de um Filtro de Bloom

- Qual a relação dos falsos positivos com os parâmetros do filtro (n e k) e do conjunto (m)?

Lançamento de m dardos para n alvos

- Caso similar ao do lançamento de m dardos para n alvos igualmente prováveis
 - por sua vez similar ao problema dos aniversários
- Se atirarmos m dados para n alvos igualmente prováveis , **qual a probabilidade de um alvo ser atingido por pelo menos um dardo ?**
- No caso dos filtros de Bloom
 - os alvos são os vários bits do filtro
 - os dardos são os valores assumidos pela função de dispersão

Probabilidade de falsos positivos ?

- Para termos falso positivo teremos de ter as k posições determinadas pelas funções de dispersão com o valor 1
 - Para elementos não pertencentes ao conjunto
- Como obter essa probabilidade?
- Comecemos pela probabilidade de um bit estar a 1 ...

Probabilidades para um bit

- Inicialmente todos os bits estão a zero
- Designando o bit na posição i por b_i ...
- Qual a probabilidade do bit $b_i = 1$ depois de aplicar a primeira função de dispersão na inserção de um elemento?
- Assumindo que a função de dispersão seleciona cada uma das posições do vetor com igual probabilidade ...
- A probabilidade é simplesmente

$$P[b_i = 1] = \frac{1}{n}$$

- pois os n alvos são equiprováveis.
- Em consequência: $P[b_i = 0] = 1 - \frac{1}{n}$

Probabilidade após aplicar k funções de dispersão

- Probabilidade de um bit se manter a zero após a inserção de um elemento ?
- É a probabilidade do bit continuar a ser zero depois de aplicadas as k funções de dispersão
- Se assumirmos que os resultados das funções de dispersão são independentes, será:

$$P[b_i = 0] = \left(1 - \frac{1}{n}\right)^k$$

Probabilidade após inserir m elementos

- Após a inserção de m elementos, assumindo independência, temos:
- $P[b_i = 0] = \left(1 - \frac{1}{n}\right)^{k m}$
- Fazendo $\left(1 - \frac{1}{n}\right)^m = a$
- $P[b_i = 0] = a^k$
- $P[b_i = 1] = 1 - a^k$
 - Relacionada com a **probabilidade de um alvo ser atingido por pelo menos um dardo**

Probabilidade de falsos positivos

- Temos um falso positivo quando temos os k bits iguais a 1 para um elemento não pertencente a C
- A probabilidade de um falso positivo, p_{fp} , após inserirmos m elementos é

$$p_{fp} = \left[1 - \left(1 - \frac{1}{n} \right)^{km} \right]^k = (1 - a^k)^k$$

- Aplicando $\lim_{n \rightarrow \infty} (1 - 1/n)^n = e^{-1}$

- Que pode ser aproximada por:

$$p_{fp} \approx (1 - e^{-km/n})^k$$

- A probabilidade de falsos positivos depende de k, m e n

Efeito de k na p_{fp}

- Exemplo:

$$m = 10^9 \text{ (mil milhões)}$$

$$n = 8 \times 10^9 \text{ (8 mil milhões)}$$

$$m/n = 1/8$$

- $k=1$

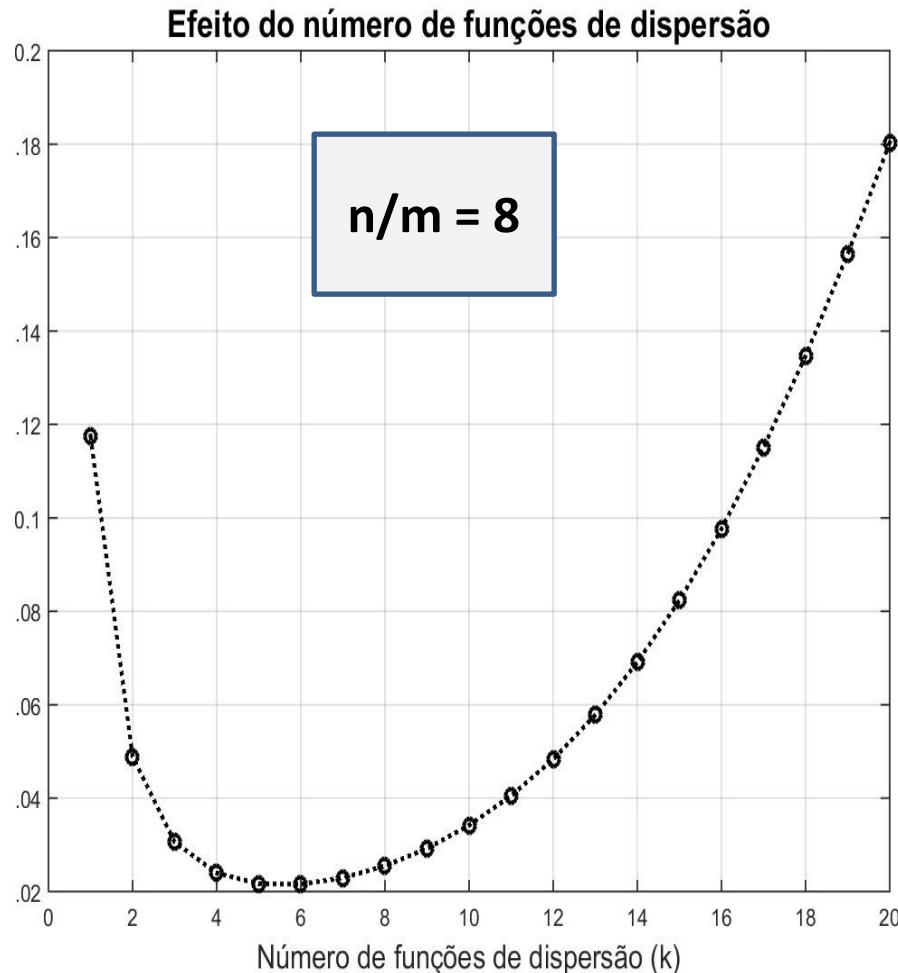
$$p_{fp} = 1 - e^{-1/8} = 0.1175$$

- $k=2$

$$p_{fp} = (1 - e^{-2/8})^2 = 0.00493$$

- Probabilidade de erro diminui com aumento de k (acontece sempre ?)
- O que acontece se formos aumentando k ?

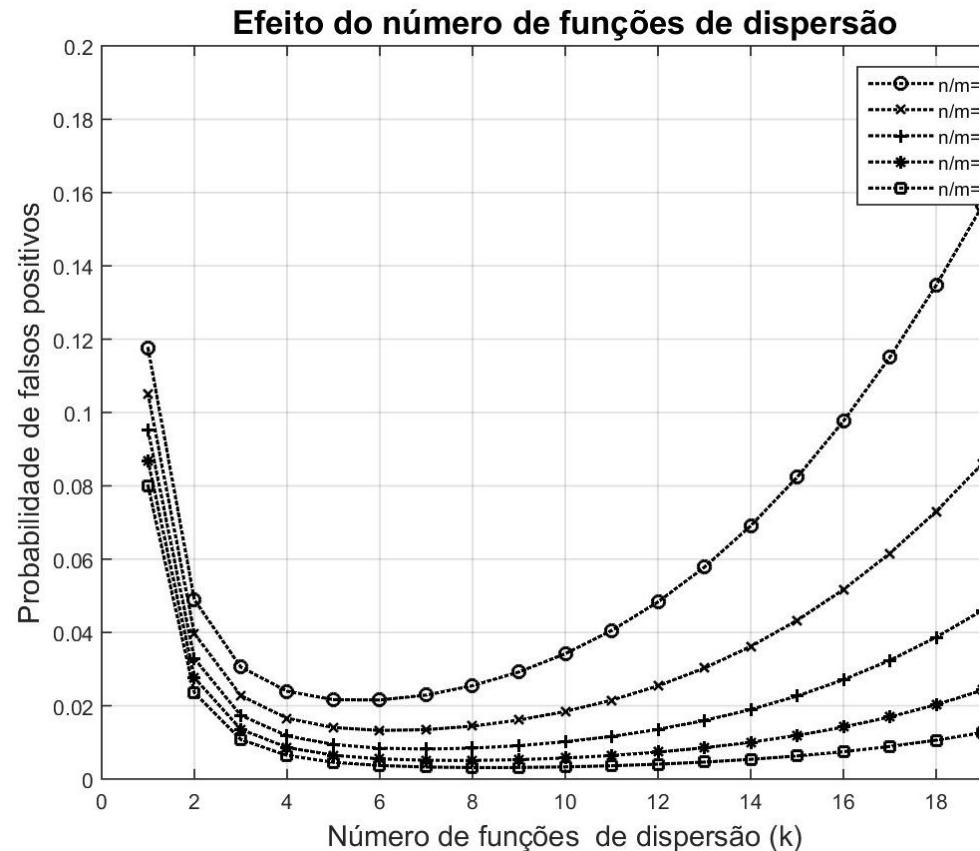
Efeito de k na prob. Falsos Positivos



- Adicionando mais funções não diminui obrigatoriamente essa probabilidade
- Decresce até um certo valor de k
- Depois aumenta
 - Motivo ?

Redução de falsos positivos

- FP podem ser reduzidos aumentando o tamanho do vetor B
 - neste caso à custa de mais memória.
 - O efeito da relação n/m é ilustrado na figura
- Também podem ser reduzidos aumentando o número de funções de dispersão
 - mas apenas até um certo valor



Então qual o valor ótimo de k ?

- É possível determinar o número de funções de dispersão que minimiza a probabilidade de falsos positivos, p_{fp}
- Para facilitar os cálculos minimiza-se $\ln(p_{fp})$
- Aplicando $\ln()$ a $p_{pf} = (1 - a^k)^k$
- temos: $\ln(p_{pf}) = k \ln(1 - a^k)$

Minimização

- Como obter k ótimo (que minimiza p) ?
- A solução usual de calcular zeros da derivada
- Derivando (em ordem a k) e igualando a zero
$$(1 - a^k) \ln(1 - a^k) - a^k \ln(a^k) = 0$$
- Que tem por solução

$$a^k = 1/2$$

[usar, por exemplo, fsolve no Matlab]

k ótimo

- O valor ótimo de *k* pode ser obtido aplicando logaritmos e resolvendo em ordem a *k*:

$$k_{\text{óptimo}} = \frac{\ln(1/2)}{\ln(a)}$$

- Substituindo o valor de *a* temos

$$k_{\text{óptimo}} = \frac{\ln(1/2)}{m \times \ln(1 - 1/n)}$$

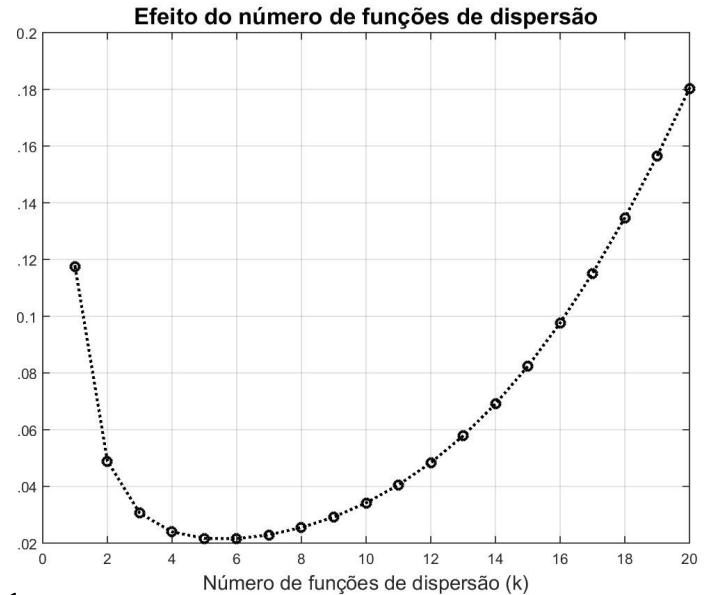
- Aproximando $\ln(1 - 1/n)$ pelo primeiro termo da série de Taylor ($-1/n$)

$$k_{\text{óptimo}} \approx \frac{n \ln(2)}{m} = \frac{0.693 n}{m}$$

- Na prática **utiliza-se o inteiro mais próximo**

Exemplo – k ótimo

- $m = 10^{12}$
- $n = 8 \times 10^{12}$
- k ótimo ?
- $k_{óptimo} \approx \frac{n \ln(2)}{m} = \frac{0.693 \times 8 \times 10^{12}}{10^{12}} = 5.54$
 - Aproximadamente 6
- Qual a relação com um dos gráficos que já vimos?



Determinação de n

- O valor de n pode ser calculado substituindo o valor ótimo de k na expressão de probabilidade

$$p_{fp} \approx (1 - e^{-km/n})^k$$

- *Sendo dados m e um objetivo em termos de probabilidade de falsos positivos p_{fp}*
- e assumindo que o valor de k ótimo é adotado

Limite inferior para a probabilidade de erro (FP)

- Se tivessemos $a^k = 1/2$ para um inteiro $k_{óptimo}$, a equação $(1 - a^k)^k$ resultaria em:

$$\begin{aligned} p_{ótima} &= \left(1 - \frac{1}{2}\right)^{k_{óptimo}} \\ &= \left(\frac{1}{2}\right)^{k_{óptimo}} \\ &= 2^{-k_{óptimo}} \end{aligned}$$

- Que pode ser considerado o limite inferior para a probabilidade de erro
 - Obviamente falsos positivos

Filtros de Bloom – Aspectos positivos

- Os filtros de Bloom garantem **não existência de falsos negativos** ...
- e usam uma **quantidade de memória limitada**
 - Ótimo para pré-processamento antes de processos mais exigentes
- Adequados para implementação em hardware
 - As funções de dispersão podem ser **parallelizadas**

Filtros de Bloom - Limitações

- Como a tabela não pode ser expandida, o máximo número de elementos a armazenar no filtro tem de ser conhecido previamente
- Assim que se excede a capacidade para a qual foi projetado, os falsos positivos aumentam rapidamente ao serem inseridos mais elementos

Filtros de Bloom - Compromissos

- Os falsos positivos podem ser diminuídos através de:
 - Aumento do número de funções de dispersão (até ao $k_{ótimo}$)
 - E do espaço alocado para armazenar o vetor

Comentários finais

- Os filtros de Bloom devem ser considerados para programas em que um teste de pertença imperfeito pode ser aplicado a um conjunto de dados (muito) grande
- As grandes vantagens de um filtro de Bloom são a rapidez e taxa de erro
- Apesar de poder ser aplicado a conjuntos de qualquer dimensão, árvores e heaps são melhores soluções para conjuntos pequenos

Filtros Bloom de contagem / Filtros de Contagem

Counting Bloom Filters

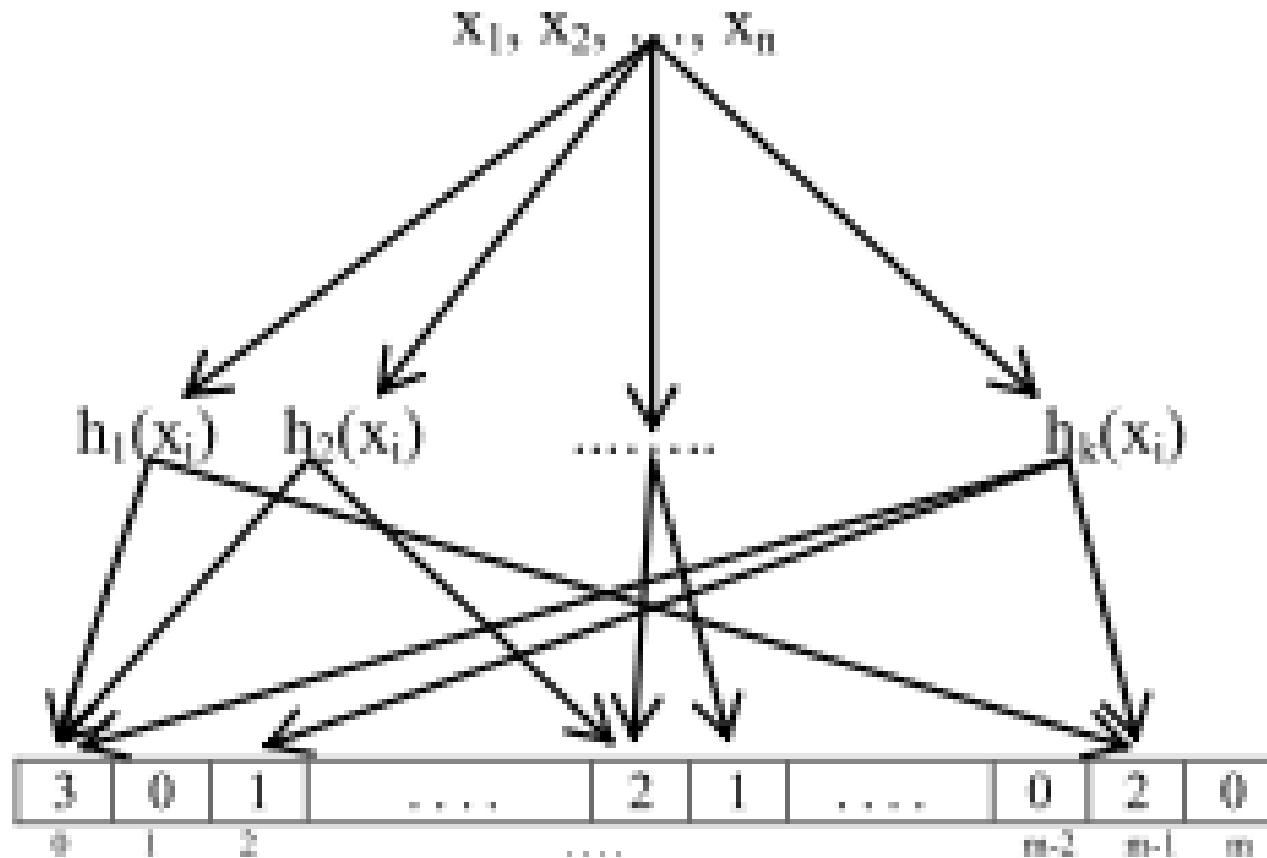
Filtros de Contagem

- Um filtro Bloom básico representa um conjunto, mas:
 - não permite a consulta da **multiplicidade** (número de vezes que elemento foi inserido)
 - nem suporta a **remoção** de elementos
- Um filtro Bloom de contagem estende um filtro Bloom básico para dar resposta a estas limitações

Filtros de Contagem

- As posições do vetor são estendidas de um único bit para um **contador de b bits**
[a versão original utilizava 4 bits]
- Na **inserção** de um elemento o **contador é incrementado**
- na **remoção** é **decrementado**
- Ocupa mais espaço
 - tipicamente 3 a 4 vezes mais que o de Bloom

Filtro de Contagem



Obtenção da multiplicidade

- Problema:
 - Os contadores correspondentes a um elemento podem também ser alterados por outros elementos
- Como ter boa estimativa do número de inserções de um elemento ?
- Solução:
 - Usar o **valor mínimo** entre os vários contadores correspondentes ao elemento

Implementação

- A implementação deste tipo de filtros em Matlab é simples
- Bastam ligeiras alterações a:
 - `adicionar()` : passa a incrementar
 - `membro()` : requer que todos sejam não nulos
- A criação da função adicional `contagem()`
- E, se necessário, a função `remover()`

contagem()

- Operação nova
- Para obter a contagem (multiplicidade) associada a um elemento do conjunto:
 1. Determina-se o conjunto de contadores que lhe correspondem
 - Através das k funções de dispersão
 2. Calcula-se o **valor mínimo** armazenado nesses contadores
 - Algoritmo como *minimum selection* (MS)

remover()

- Implementação similar a adicionar()
- Esta operação introduz a possibilidade de falsos negativos
 - Quando se coloca um bit a zero que era parte dos k bits de outro elemento, o filtro deixará de o considerar como pertencendo ao conjunto

Problemas ...

- *Overflow do contador*
 - Quando a contagem chega a $2^b - 1$
 - Tipicamente as implementações param de contar
 - Preferível a recomeçar em 0 (introduziria falsos negativos)
 - Mas introduz mais erro na estimativa da contagem
- *Escolha de b* (número de bits dos contadores)
 - Um valor grande reduz a poupança de espaço
 - Um valor pequeno rapidamente leva a overflow
 - Escolha do valor é um compromisso e depende dos dados

Outras Técnicas

- Variantes do Filtro de Bloom
 - Ver, por exemplo:
 - <http://matthias.vallentin.net/course-work/cs270-s11.pdf>
- Cuckoo Hashing
 - Ver:
 - [http://www.lkozma.net/cuckoo hashing visualization/](http://www.lkozma.net/cuckoo_hashing_visualization/)

Note to other teachers and users of these slides: We would be delighted if you found this our material useful in giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: <http://www.mmds.org>

PPT baseado em: Mining Data Streams

Mining of Massive Datasets
Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University
<http://www.mmds.org>



Outras fontes utilizadas

- <http://matthias.vallentin.net/blog/2011/06/a-garden-variety-of-bloom-filters/>
- [https://en.wikipedia.org/wiki/Bloom_filter#Counting filters](https://en.wikipedia.org/wiki/Bloom_filter#Counting_filters)

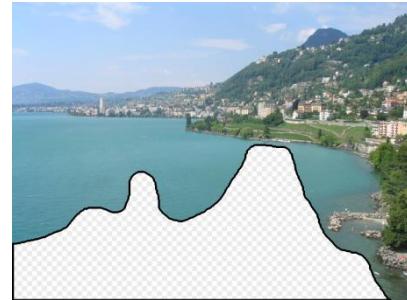
MPEI

Solução Probabilística
para a Procura de Similares

Problema Exemplo – Completar cena

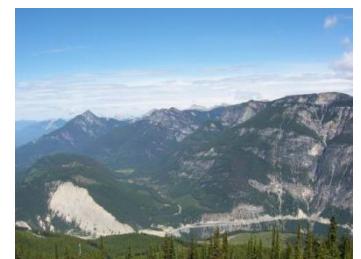
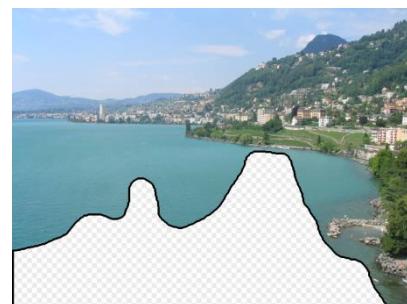


Problema Exemplo – Completar cena



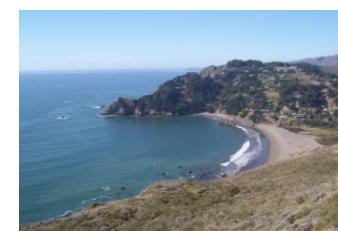
Problema Exemplo – Completar cena

- 10 imagens mais próximas numa coleção de 20 000 imagens



Problema Exemplo – Completar cena

10 imagens mais próximas num conjunto de 2 milhões de imagens



Generalizando

- Muitos problemas podem ser expressos em termos da **descoberta de conjuntos similares**
- Exemplos:
 - Páginas web similares
 - para deteção de potenciais cópias
 - Clientes que compraram produtos similares
 - Imagens com características similares
 - Utilizadores que visitam sites similares
 - Clientes que compraram livros similares

Generalizando (continuação)

- Em todos estes problemas temos **entidades** que podem ser **representadas por um conjunto**
 - páginas web, compras, imagens, utilizadores, ...
- Exemplo:
 - As **páginas web** podem ser representadas pelo **conjunto das palavras** que contêm

Definição do Problema

- Tendo:
- **pontos** x_1, x_2, \dots num espaço com n dimensões
 - Exemplo: imagem é um vetor com as cores dos pixels
$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \rightarrow [1\ 2\ 1\ 0\ 2\ 1\ 0\ 1\ 0]$$
- uma função de **distância** $d(x_1, x_2)$
 - Que quantifica a distância entre x_1 e x_2
- **Objectivo:** Determinar todos os pares de dados (x_i, x_j) com distância igual ou inferior a um determinado limiar s , $d(x_i, x_j) \leq s$

Solução ingénua

- **Comparar todos os pares** possíveis
- Com N pontos, teríamos complexidade $O(N^2)$
- Muito demorada ou mesmo impossível em tempo útil para N grande
- Exemplo:
 - 1 milhão de documentos ($N = 10^6$)
 - temos de calcular a similaridade para cada par
 - $N(N - 1)/2 = 5 \times 10^{11}$ comparações
 - Com 86400 s/dia e 10^6 comparações/s, levaria mais de 5 dias
 - Se tivermos 10 milhões é muito pior, demora mais de um ano.

Distância

- O objectivo é determinar os vizinhos mais próximos no espaço n -dimensional
- Formalmente vizinhos próximos (near neighbors) são pontos que se encontram a uma “pequena distância”
- Em primeiro lugar tem de se definir o que significa distância

Distância e Similaridade de Jaccard

- A **similaridade** (ou semelhança) **de Jaccard** de 2 **conjuntos** é definida pelo quociente entre a dimensão da sua **interseção** e dimensão da sua **união**:

$$sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$

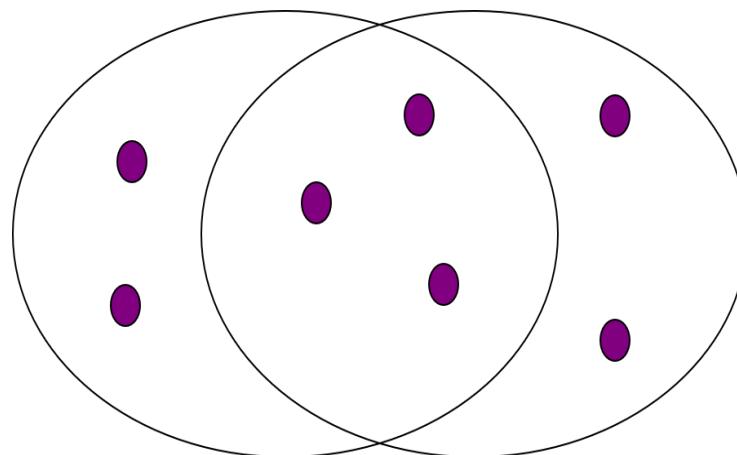
– representando || o nº de elementos do conjunto ou cardinalidade do conjunto

- A **distância Jaccard**, d_J , é obtida diretamente da similaridade:

$$d_J(C_1, C_2) = 1 - |C_1 \cap C_2| / |C_1 \cup C_2|$$

Exemplo

- Qual a distância de Jaccard entre os 2 conjuntos?



- Resolução
 - Temos 3 elementos na interseção e 7 na união ...
 - $d_J = 1 - 3/7 = 4/7$

Exemplo em Matlab

- Qual a similaridade entre as seguintes strings
- `str1='When nine hundred years old you reach, look as good you will not.'`
- `str2='You will not look as good when nine hundred years old'`
- `C1=unique(strsplit(lower(str1)));`
- `C2=unique(strsplit(lower(str2)));`
- `simJ=length(intersect(C1,C2))/ length(union(C1,C2))`
- Resultado: `simJ = 0.7692`

Aplicação Exemplo #1 (Matlab)

- Detetar textos similares
- Exemplo para demonstração (toy example)
- Conjuntos serão as palavras (únicas) dos textos
 - Sem pós-processamento
- Aplicação direta da distância de Jaccard

Aplicação Exemplo #1

Tarefas principais

1. Criar Conjuntos com as palavras de todos os documentos

```
Sets{1}=getSetOfWordsFromFile('texto1.txt')
```

```
Sets{2}=getSetOfWordsFromFile('texto2.txt')
```

...

2. Calcular a distância de Jaccard para todos os pares

```
distJ=calcDistancesJ(Sets);
```

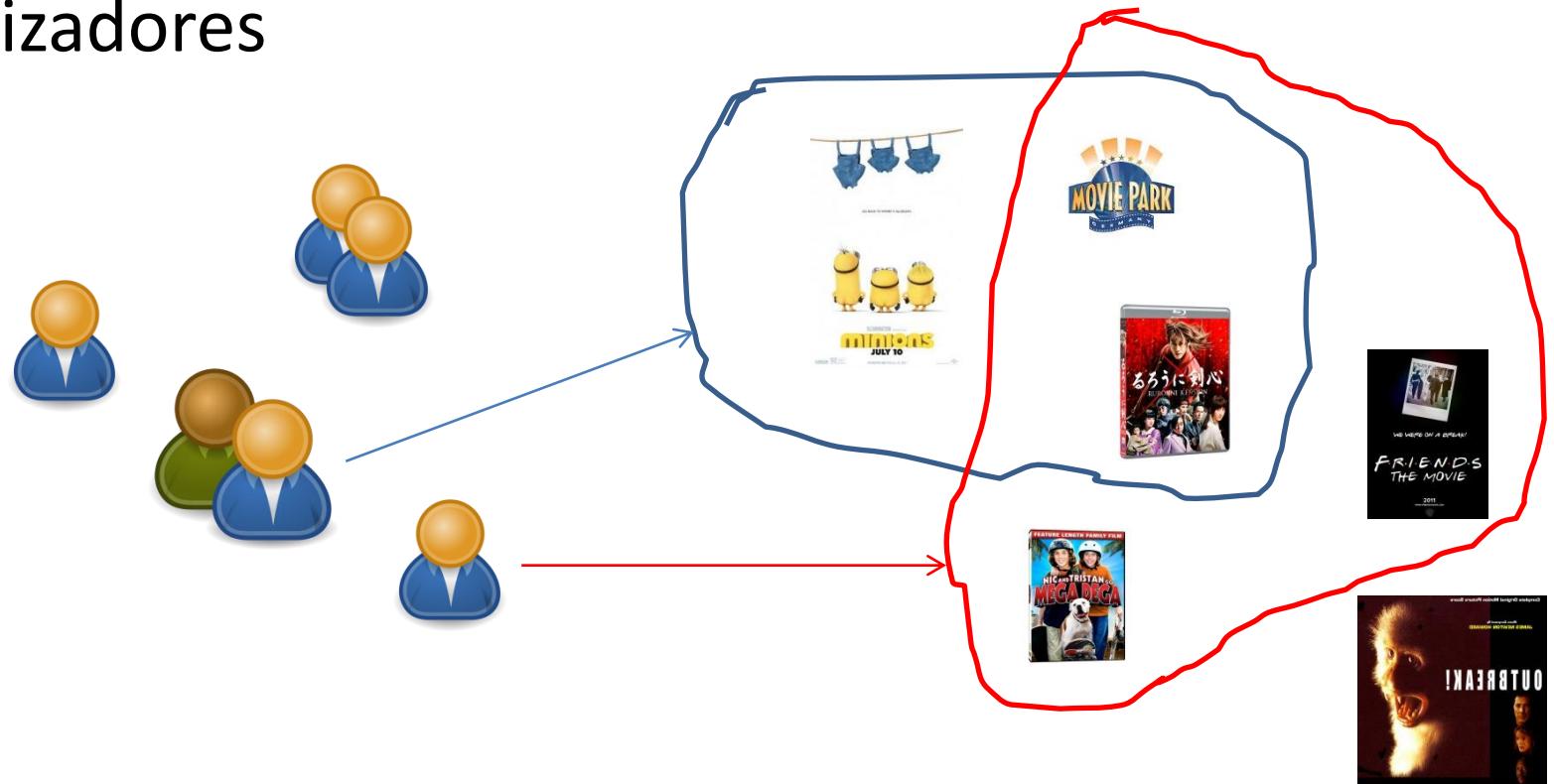
3. Determinar os pares que têm distância inferior a um certo limiar

```
Similar=findSimilar(distJ,threshold,ids);
```

4. Mostrar resultados

Aplicação Exemplo #2

- Determinar **conjuntos de filmes** avaliados por utilizadores



- Assunto de um dos Guiões Práticos

MovieLens

- MovieLens
(<http://movielens.org>)
é um site que ajuda na
selecção de filmes a ver
 - Tem muitos utilizadores
registados



The image displays three sections of the MovieLens website:

- recommendations**: A section titled "Based on your ratings, MovieLens recommends these movies" featuring "Band of Brothers" (2001), "Casablanca" (1942), "One Flew Over the Cuckoo's Nest" (1975), "The Lives of Others" (2006), "Sunset Boulevard" (1950), "The Third Man" (1949), and "Patt" (1957).
- top picks**: A section titled "see more" featuring "Band of Brothers" (2001), "Casablanca" (1942), "One Flew Over the Cuckoo's Nest" (1975), "The Lives of Others" (2006), "Sunset Boulevard" (1950), "The Third Man" (1949), and "Patt" (1957).
- recent releases**: A section titled "see more" featuring "Cartelitas" (2014), "Felony" (2014), "What If" (2014), "Frank" (2014), "Sin City: A Dame to Kill For" (2014), "If I Stay" (2014), and "Aren't You Gonna Miss Me" (2014).

Conjuntos de dados MovieLens

- GroupLens Research criou e disponibilizou **conjuntos de dados** (data sets) do site MovieLens
 - Disponíveis em: <http://grouplens.org/datasets/movielens/>
- Os dados foram recolhidos ao longo de diferentes períodos de tempo
- Existem vários conjuntos, de vários tamanhos:
 - **MovieLens 100K Dataset**
 - MovieLens 1M Dataset
 - ...

Conjunto de dados MovieLens 100K

- Conjunto de dados estável e usado como padrão (benchmark)
 - Disponibilizado em 4/1998
- Aprox. 100 000 avaliações
 - de 943 utilizadores, sobre 1682 filmes
- [README](#)
- Link: <http://grouplens.org/datasets/movielens/100k/>

Ficheiro u.data

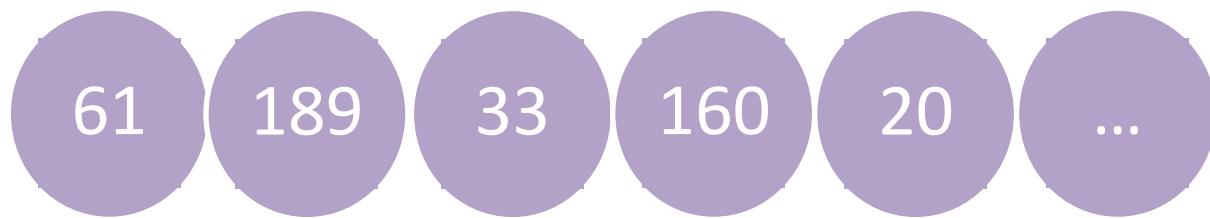
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467
305	451	3	886324817

...

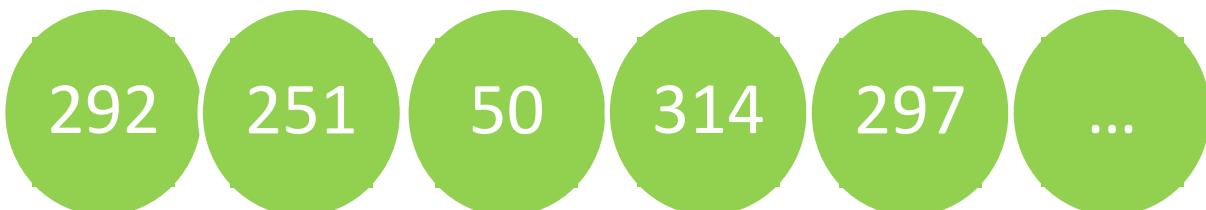
- 1^a coluna contém o ID do utilizador
- 2^a coluna o ID do filme
 - avaliado pelo utilizador da 1^a coluna
- Avaliação na 3^a coluna
- 4^a coluna é Informação de tempo (timestamp)

Conjuntos de filmes (avaliados por cada utilizador)

Utiliz. 1

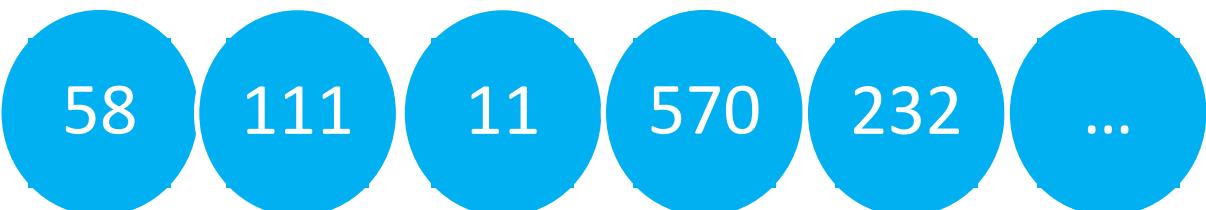


Utiliz. 2



...

Utiliz.
943



Demonstração

- Primeira “solução”
 - Utilização direta da distância de Jaccard
- Muito lento 😞

Resultado

- Results (similar sets):

328 788 distance = 0.327

408 898 distance = 0.161

489 587 distance = 0.370

Conjuntos Grandes e Gigantes

- **Objetivo:**

**Dado um grande número de documentos (N),
determinar pares “quase iguais”**

– $N = \text{milhões, milhares de milhões, biliões, ...}$

Conjuntos Grandes e Gigantes

- **Problemas:**
- Demasiados documentos para se compararem todos os pares
- Muitas partes de um documento podem aparecer por outra ordem noutra
- Documentos são tão grandes ou número elevado que não cabem em memória

Conjuntos Grandes e Gigantes

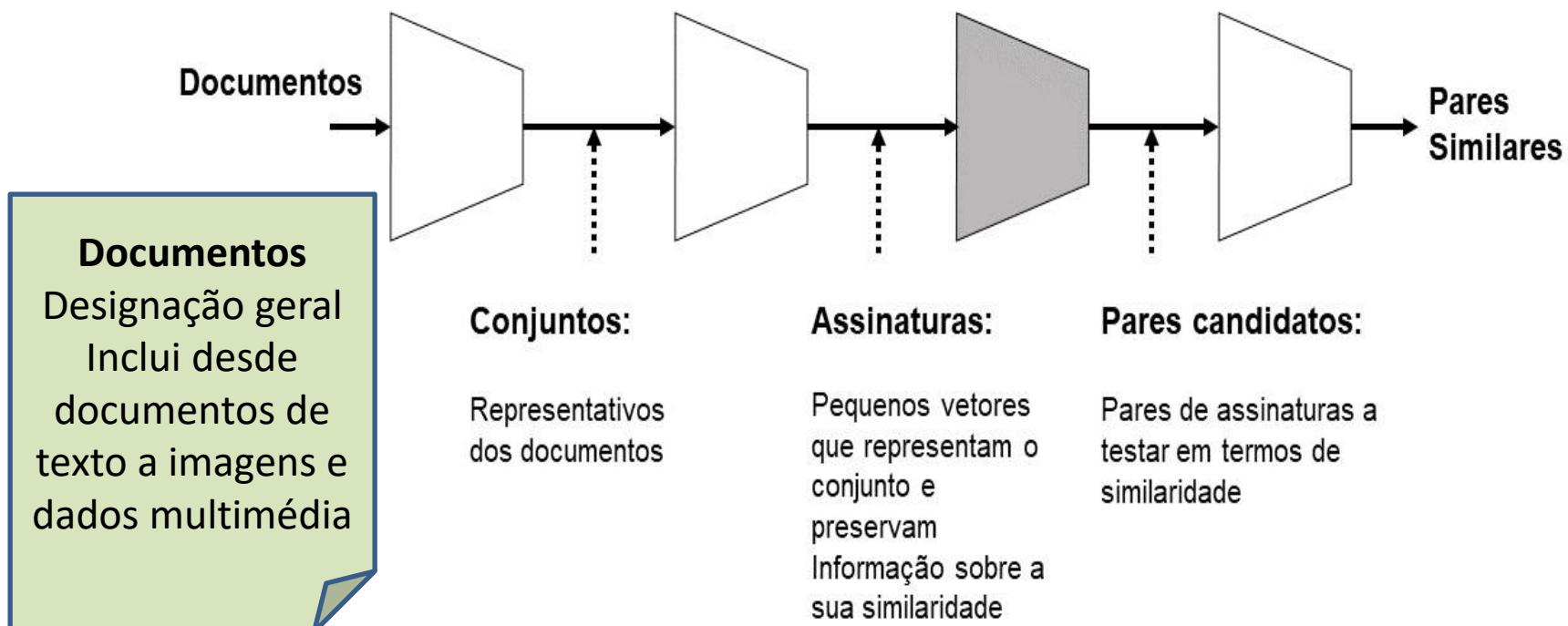
Solução

- Para N grande ou muito grande a solução passa por:
 1. **Reducir a dimensão dos conjuntos**
 - mantendo a informação essencial à determinação de distância entre eles
 2. **Reducir o tempo de cálculo da distância e/ou reduzir os pares** a que se tem de aplicar essa distância.
- A resolução destes problemas é habitualmente feita em sequência

Abordagem probabilística

Visão geral

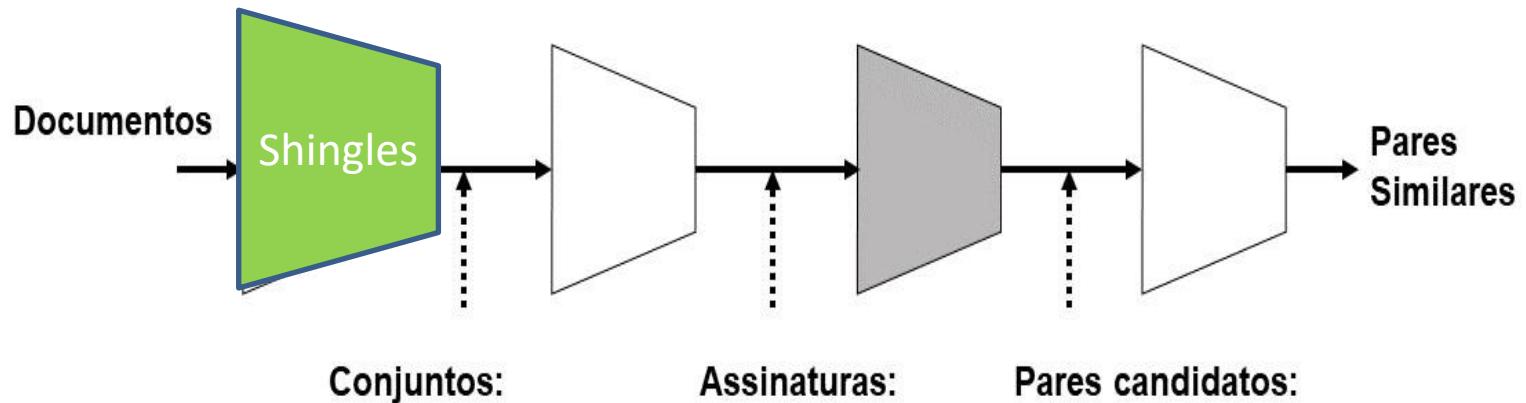
- Processo de determinação de documentos similares:



Processo

- O processo consiste nas seguintes etapas sequenciais:
 1. **Obtenção dos conjuntos representativos**
 2. **Redução desses conjuntos** a conjuntos de dimensão fixa e pequena
O resultado é usualmente designado por **assinatura**
 3. (opcional) Processamento das assinaturas por forma a identificar pares potencialmente similares
 4. **Cálculo de similaridade** dos pares de conjuntos
 - todos ou os resultantes do passo anterior

1 - Conversão dos documentos em conjuntos



Criação dos conjuntos representativos

- O objetivo desta primeira etapa é **criar os conjuntos representativos dos documentos**
 - A informação relevante a reter depende obviamente do tipo de documento
- Sem perda de generalidade, **consideraremos documentos constituídos por palavras**
 - ou para sermos mais precisos, sequências de caracteres
- A aplicação a outro tipo de documentos pode fazer-se adaptando o apresentado para sequências de caracteres
 - Por exemplo, no caso de imagens pode considerar-se como equivalente à palavra o valor de cada pixel (valor inteiro ou triplo RGB)

Soluções

- As soluções mais simples são:
 1. conjunto de palavras que ocorrem no documento
 2. conjunto das palavras “importantes”.
- Ambas sofrem do mesmo problema:
 - não preservam informação sobre a ordem de ocorrência
- A ordem de ocorrência pode ser tida em conta utilizando sequências de palavras (ou de caracteres), ideia na base dos ***k-gramas***
 - também conhecidos por *k-shingles*
 - ou simplesmente ***Shingles***

Shingles

- Um k -shingle (ou k -grama) para um documento é uma **sequência de k símbolos** que aparecem no documento
- Os símbolos podem ser caracteres, palavras ou outra informação, dependendo da aplicação
 - ex: código de cor de um pixel
- Assume-se que documentos que têm muitos *Shingles* em comum são semelhantes
- Utilizando *Shingles*, um documento D é representado pelo conjunto dos seus k -gramas $C = S(D)$

Exemplo

- Quais os Shingles do documento contendo a sequência de caracteres ‘**abcab**’ considerando $k=2$
- Conjunto se 2-shingles: $S(D) = \{ab, bc, ca\}$
- Se aceitamos repetições: $S'(D)=\{ab, bc, ca, ab\}$
 - ab aparece 2 vezes

Similaridade para Shingles

- Representando um documento D_i pelo seu conjunto de k-shingles $C_i = S(D_i)$
- Uma medida natural de similaridade é a similaridade de Jaccard
 - calculada com base nos conjuntos de Shingles representativos dos documentos

$$\text{sim}(D_1, D_2) = \text{sim}(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$

Escolha de k

- A escolha de k não é trivial
- Deve escolher-se k suficientemente grande para evitar que a maioria dos documentos tenha a maioria dos *Shingles*
 - evitando desta forma que a generalidade dos documentos sejam representados pelo mesmo conjunto.
- Na prática:
 - $k = 5$ é bom para documentos curtos
 - e $k = 10$ é mais adequado para documentos longos.

Representação binária

- Para simplificar cálculo de interseção e união, os documentos podem ser representado por um vetor de zeros e uns no espaço de k-gramas (vetor binário)
 - em que cada Shingle/k-grama é uma dimensão
- Nesta representação a interseção e união são operações de bits (AND e OR)
- Os vetores de um conjunto de documentos formam uma matriz

Exemplo

- 4 documentos:
 - $D_1 = 'aab'$
 - $D_2 = 'bcd'$
 - $D_3 = 'cda'$
 - $D_4 = 'cd'$
- 2-shingles (sem repetição) existentes nos 4 documentos:
 - $S(D) = \{aa, ab, bc, cd, da\}$
- Usando as linhas para os diferentes shingles e pela ordem em $S(D)$ temos a matriz:

	D1	D2	D3	D4
aa	1	0	0	0
ab	1	0	0	0
bc	0	1	0	0
cd	0	1	1	1
da	0	0	1	0

Exemplo (continuação)

- $d(D_2, D_3) = ?$
- $= d(C_2, C_3) = ?$
- $C_2 = 00110$
- $C_3 = 00011$
- $|Interseção| = 1$
 - Bitwise AND
- $|União| = 3$
 - Bitwise OR
- **Sim Jaccard = 1/3**
- $d(C_2, C_3) = 1 - \text{simJ} = 2/3$

aa	1	0	0	0
ab	1	0	0	0
bc	0	1	0	0
cd	0	1	1	1
da	0	0	1	0

D1 D2 D3 D4

Outro Exemplo

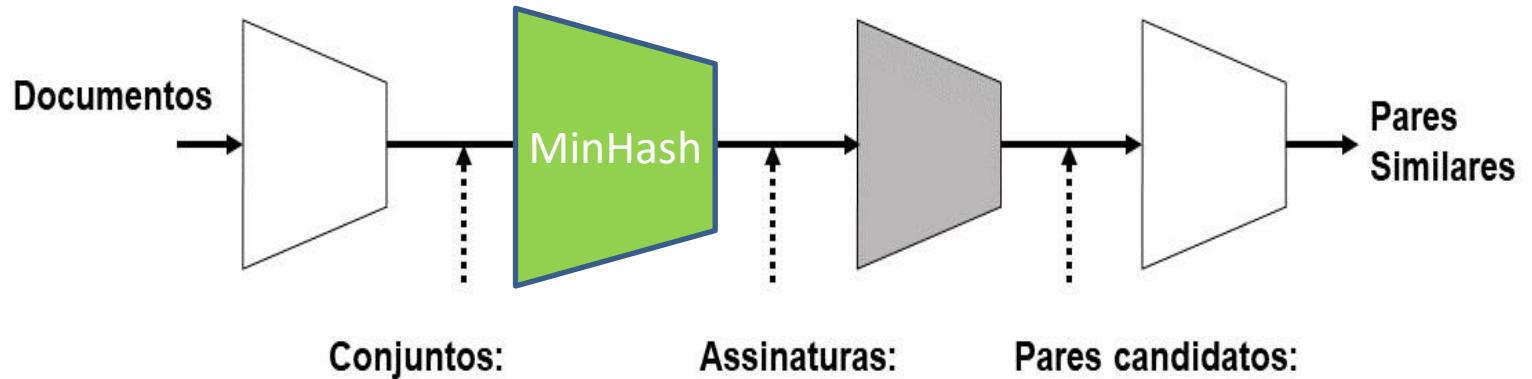
- **Cada documento é uma coluna**
 - C_i representa o Documento i
- **Exemplo:** $\text{sim}(C_1, C_2) = ?$
- Comprimento da **interseção** = 3
- Comprimento da **união** = 6
- Similaridade de Jaccard similarity (não é a distância) = 3/6
- $d(C_1, C_2) = 1 - (\text{similaridade Jaccard}) = 3/6$

	Documentos			
Shingles	1	1	1	0
→	1	1	0	1
→	0	1	0	1
→	0	0	0	1
→	1	0	0	1
→	1	1	1	0
→	1	0	1	0

A seguir: Descobrir Colunas Similares

- **Até agora:**
 - Documentos → Conjuntos de Shingles
 - Representação dos conjuntos por vetores booleanos numa matriz
- **Próximo objectivo: Descobrir colunas similares usando representações compactas (assinaturas)**

2 - Cálculo das Assinaturas



Assinaturas

- Uma das etapas importantes do processo é a redução da representação dos conjuntos
- Ideia base:
- Mapear cada conjunto C_i para uma pequena assinatura $Sig(C_i)$ através de funções rápidas, tal que:
 1. $Sig(C)$ é suficientemente pequena para que a assinatura de um número muito grande de conjuntos possa ser mantida em memória RAM
 2. A similaridade das assinaturas $Sig(C_1)$ e $Sig(C_2)$ é aproximadamente igual à $sim(C_1, C_2)$

Desafio

- Obter uma função de dispersão $h()$ tal que:
 - Se $\text{sim}(C_1, C_2)$ é elevada, então com elevada probabilidade $h(C_1) = h(C_2)$
 - Se $\text{sim}(C_1, C_2)$ é baixa, então $h(C_1) \neq h(C_2)$ com elevada probabilidade
- A função $h()$ depende da métrica de similaridade
- Para a similaridade de Jaccard **a função Min-Hash cumpre os requisitos**

Redução do conjunto usando permutações

- Uma forma de reduzir o conjunto C representativo de um documento é considerar apenas um subconjunto
- A seleção pode ser feita usando **permutações aleatórias**:
 - Aplicação de uma permutação aleatória π às linhas da matriz booleana
 - Reter o valor do índice da primeira linha (na ordem permutada) correspondente a um Shingle (ou equivalente) existente

Mínimo da função

- Esta operação pode ser vista como a aplicação de uma **função de dispersão** $h_\pi(C) = \text{índice da primeira linha}$ (na ordem permutada) **na qual a coluna C tem valor 1**

$$h_\pi(C) = \min_\pi \pi(C)$$

- Repetir para várias permutações independentes, por exemplo 100, para obter um vetor (assinatura)
 - Este processo de repetição com permutações independentes pode ser visto como a aplicação de várias funções $h_\pi()$

Exemplo

2º elemento da permutação é o primeiro a mapear para um 1

Permutação π

2	4	3
3	2	4
7	1	7
6	3	2
1	6	6
5	7	1
4	5	5

Matriz de Entrada (Shingles x Documentos)

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

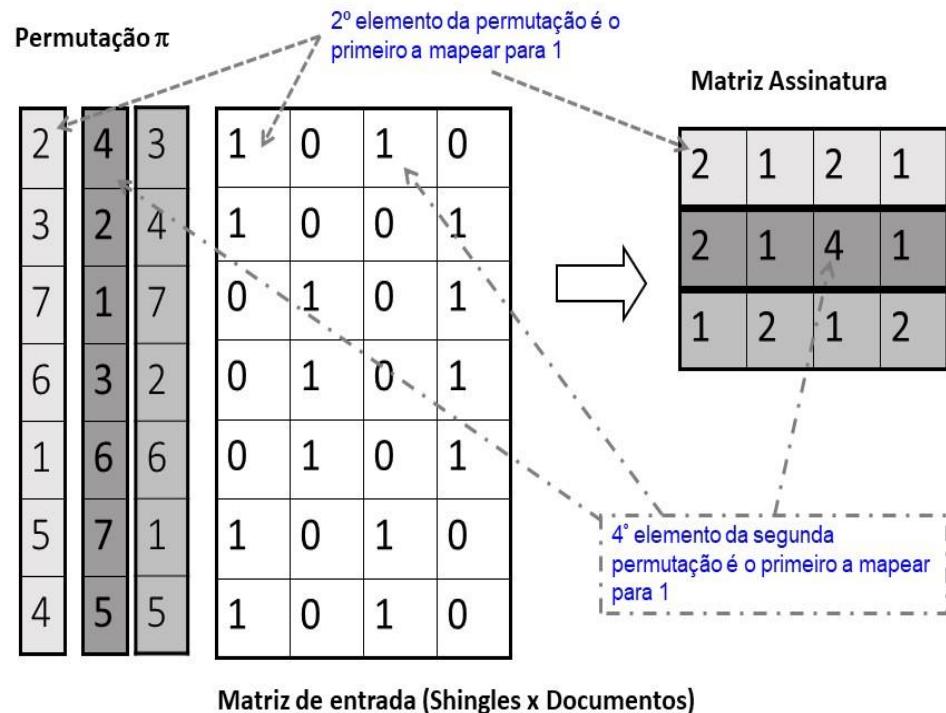
Matriz de Assinaturas M

2	1	2	1
2	1	4	1
1	2	1	2

4º elemento da permutação é o primeiro a mapear para um 1

Exemplo (continuação)

- 4 documentos representados num espaço de 7 *Shingles* diferentes
- Na primeira permutação, a primeira linha da matriz será a quinta, a segunda a primeira, etc.
- Em consequência, na primeira permutação, o valor a considerar para o primeiro documento será 2, primeira linha da permutação com 1 na primeira coluna da matriz dos documentos
- De forma similar se obtém o resto da primeira linha da matriz assinatura
- As outras 2 linhas da matriz assinatura são obtidas usando as outras duas permutações



Propriedade de h_π

- A função $h_\pi()$ permite obter assinaturas pois estas mantêm informação sobre a similaridade dos documentos devido à seguinte propriedade
- Para uma permutação aleatória, π

$$P [h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$$

“Demonstração”

- Considere-se o documento D (conjunto de *shingles*), e $s \in D$ um shingle
- Como é igualmente provável que qualquer $s \in D$ seja mapeado para o mínimo
- Então: $P[\pi(s) = \min(\pi(D))] = 1/|D|$

“Demonstração” (cont.)

- O universo dos valores de $\min(\pi(C_1 \cup C_2))$ é $|C_1 \cup C_2|$
- Em consequência temos $|C_1 \cup C_2|$ casos possíveis
 - Todos os Shingles podem ser o mínimo
- Por outro lado, em $|C_1 \cap C_2|$ casos temos $\min(\pi(C_1)) = \min(\pi(C_2))$
 - Ou seja $|C_1 \cap C_2|$ casos favoráveis
- Tendo em conta a equiprobabilidade, o que interessa é a dimensão dos conjuntos, temos:
$$\Pr[\min(\pi(C_1)) = \min(\pi(C_2))] = |C_1 \cap C_2| / |C_1 \cup C_2|$$
 - A Similaridade de Jaccard

Propriedade de h_π

- Para duas colunas A and B:
- $h(A) = h(B)$ quando o mínimo da função de hash faz parte da interseção $A \cap B$.

A	B
0	0
0	0
1	1
0	0
0	1
1	0

Confirmação por Simulação

- A propriedade do Min-Hash anterior pode ser comprovada através de simulação

```
%% criar dois conjuntos C1 e C2 (colunas de bits)
limiar1=rand(); limiar2=rand();
C1=rand(NS,1)>limiar1;
C2=rand(NS,1)>limiar2;

%% calcular distância de Jaccard
% (com operações com bits)

I= C1 & C2; % interset
U= C1 | C2;
dJaccard=1-(sum(I) / sum(U));
fprintf(1, 'distância Jaccard=% .4f\n', dJaccard);
```

Confirmação por Simulação

```
%% estimar com permutações
EXPERIMENTS=10000; cfav=0; p=zeros(1,EXPERIMENTS);

for i=1:EXPERIMENTS
    % permutar
    pi1=C1(randperm(NS)); pi2=C2(randperm(NS));

    % mínimos
    min1=min(find(pi1>0)); min2=min(find(pi2>0));

    % se mínimos iguais
    if (min1==min2) cfav=cfav+1; end
    p(i)=cfav/i;
end

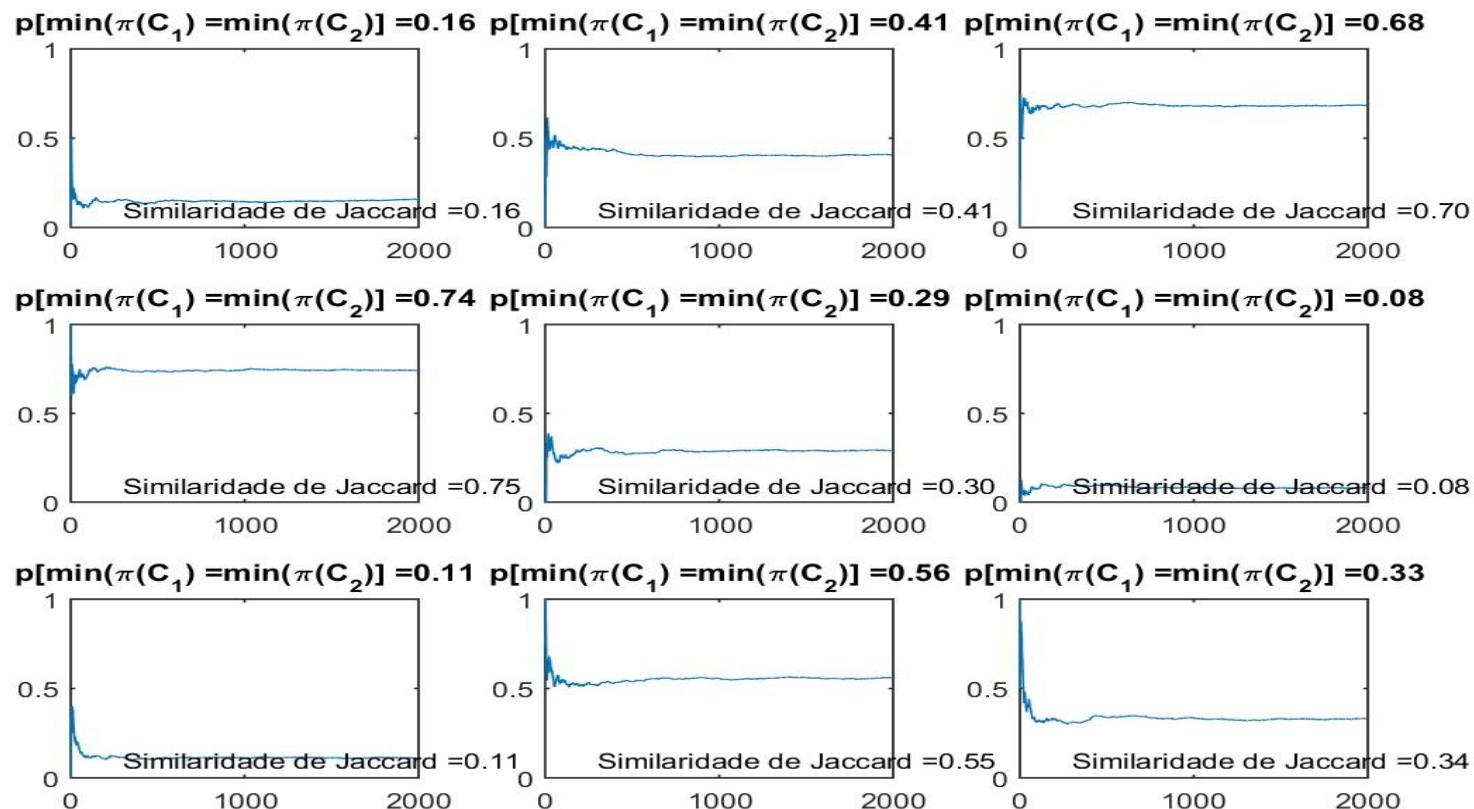
figure(1);
plot(1:i,p(1:i)); title(['p[min(\pi(C_1)) =min(\pi(C_2))] =' ...
                           sprintf('%.3f',p(i))]);

sim=cfav/EXPERIMENTS; distancia=1-sim;

fprintf(1,'Distância por simulação=%.4f\n',distancia);
```

Confirmação por Simulação

- Exemplo de resultados



Assinaturas

- Seleccione **k permutações aleatórias das linhas**
- Pense na assinatura $\text{sig}(\mathbf{C})$ como um vetor
- $\text{sig}(\mathbf{C})[i] = \text{índice da primeira linha}$ da coluna C que contém 1 de quando aplicada a permutação i a essa linha
$$\text{sig}(\mathbf{C})[i] = \min (\pi_i(\mathbf{C}))$$
- **Notq:** A **assinatura de um documento é pequena**
 - ~100 bytes para k=100 ☺
- **Atingimos um dos nossos objetivos!**
 - “Comprimimos” vetores de bits longos em pequenas assinaturas

Similaridade das Assinaturas

- Vimos que: $\Pr[h_\pi(C_1) = h_\pi(C_2)] = \text{sim}(C_1, C_2)$
- Generalizando para várias funções
 - A **similaridade de 2 assinaturas** é a **fração de funções em que concordam**
 - **Nota:** Devido à propriedade de h_π , a similaridade das colunas é a mesma que a similaridade esperada das suas assinaturas

Exemplo de aplicação

Permutações π

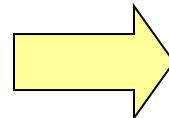
2	4	3
3	2	4
7	1	7
6	3	2
1	6	6
5	7	1
4	5	5

Matriz (Shingles x Documentos)

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Matriz de assinaturas M

2	1	2	1
2	1	4	1
1	2	1	2



Similaridades:

Col/Col
Assin

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Assin	0.67	1.00	0	0

Na prática...

- Permutar linhas para conjuntos de dimensão elevada é demorado e em geral **proibitivo**
- No entanto, o que necessitamos é de uma função que determine os índices e calcular o seu mínimo
- Uma solução eficiente **é utilizar uma função de dispersão**
 - evitando ao máximo colisões
 - O mapeamento efetuado pela função dá-nos a permutação aleatória
- Como a solução consiste na determinação **do mínimo dos valores de uma função de dispersão (hash)** é conhecido por **Min-Hash**

A propriedade mantém-se ?

- Mantemos a propriedade usando as funções de dispersão?
- Sim.
- Seja $h()$ essa função de dispersão que mapeia os membros de C_1 e C_2 para inteiros distintos
- Seja $h_{min}(C)$ o membro x de C com o valor mínimo
- Se aplicarmos $h_{min}()$ a C_1 e C_2 , obteremos o mesmo valor exatamente quando o elemento da união $C_1 \cup C_2$ com valor mínimo da função de dispersão estiver na intersecção $C_1 \cap C_2$
- A probabilidade disso ser verdade é
$$|C_1 \cap C_2| / |C_1 \cup C_2|$$
- e portanto:

$$P[h_{min}(C_1) = h_{min}(C_2)] = sim_J(C_1, C_2)$$

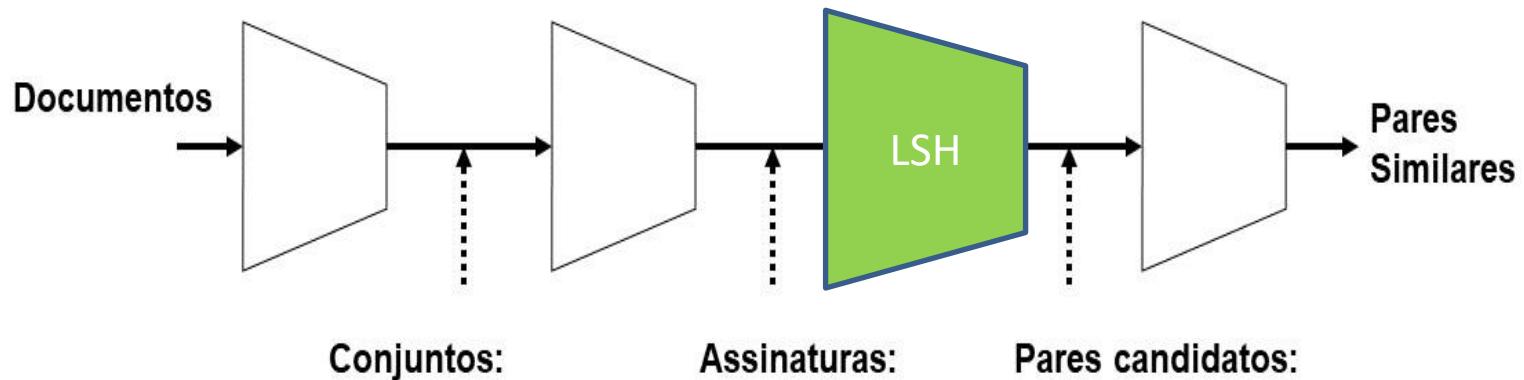
Múltiplas funções de dispersão

- Definindo a variável aleatória X como um quando $h_{min}(C_1) = h_{min}(C_2)$ e como zero caso contrário, então X é um estimador não enviesado (unbiased) da similaridade de Jaccard
- Por assumir apenas os valores zero ou um, X tem uma variância muito elevada para ser útil na prática
- Como reduzir a variância?
- Utilizando várias variáveis construídas da mesma forma
- A estimativa para a distância é igual à fração de funções de dispersão que concordam

Aplicação exemplo

- Não está convencido?
- Apliquemos ao exemplo das avaliações de filmes (MovieLens)
 - Estimando a similaridade usando o mínimo dos hash codes

3 - Determinação de pares candidatos



Pares candidatos

- Mesmo com a redução drástica obtida com as assinaturas, ter de **comparar todos os pares é algo que tem de ser evitado**
- Precisamos de **reduzir o número de pares a comparar**
- O nosso objetivo é encontrar documentos com similaridade, de Jaccard, superior a um determinado limiar
 - por exemplo, $s = 0.8$ (ou distância inferior a 0.20).

Candidatos Segundo o MinHash

- **Selecionar um limiar de semelhança s ($0 < s < 1$)**
- As colunas x e y da matriz de assinaturas são um **par candidato** se as suas assinaturas concordarem em pelo menos uma fração s das suas linhas
 $M(i, x) = M(i, y)$ para pelo menos a fração s valores de i
- Espera-se que os documentos x e y tenham a mesma similaridade (de Jaccard) que as suas assinatura

Ideia base do método LSH (*Locality-Sensitive Hashing*)

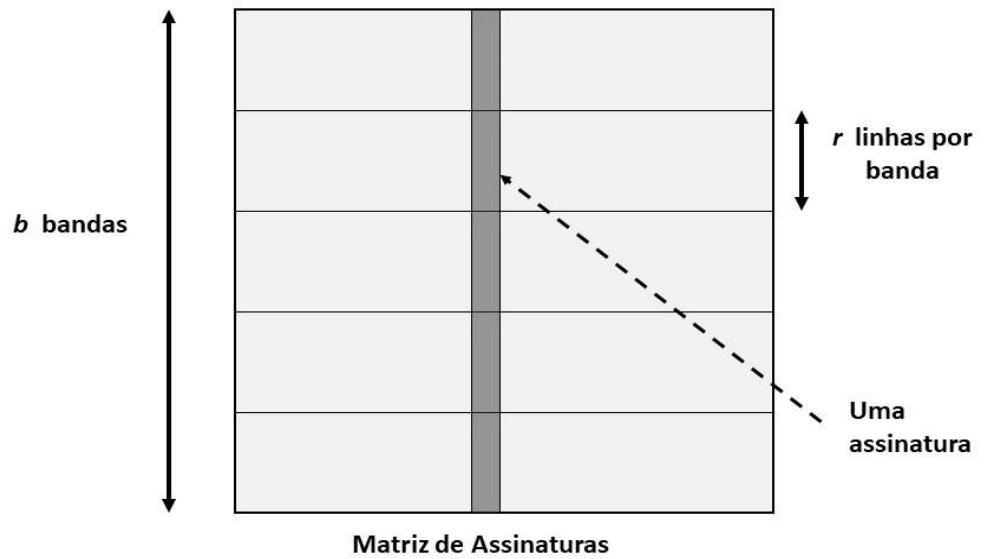
- **Objectivo:** Encontrar documentos com similaridade de Jaccard de pelo menos s
 - Para um determinado limiar, por exemplo $s = 0.8$.
- **Ideia base:**
 - Utilizar uma função de dispersão $f(x,y)$ que indica se x e y constituem um par candidato
 - Par de elementos cuja similaridade tem de ser avaliada
 - Aplicar a função às colunas da matriz de assinaturas
 - Cada par de colunas que resultam no mesmo valor da função de dispersão é um par candidato

LSH aplicado à matriz de MinHash

- Grande ideia: Aplicar funções de dispersão às colunas da matriz várias vezes
- Fazer com que (apenas) colunas similares tenham elevada probabilidade de terem o mesmo hash code
- Pares candidatos são aqueles que resultam no mesmo hash code

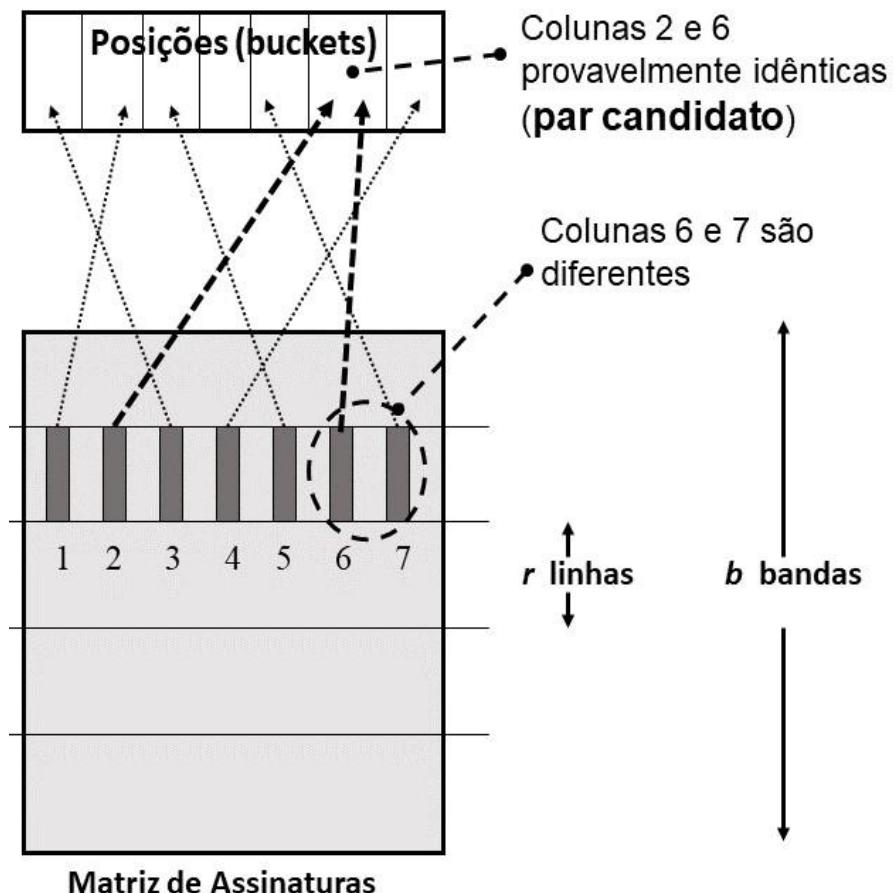
LSH na prática

- Na prática aplica-se a cada coluna **várias funções de dispersão**
- Divide-se a matriz de assinaturas em ***b* bandas**
 - de r linhas



LSH na prática (continuação)

- Aplica-se a função de dispersão a cada banda
 - Que mapeia numa de k posições
 - com k suficientemente grande
- **Pares candidatos** são mapeados para a **mesma posição** pela função de dispersão para pelo menos uma das bandas
 - No nosso exemplo: (2,6)



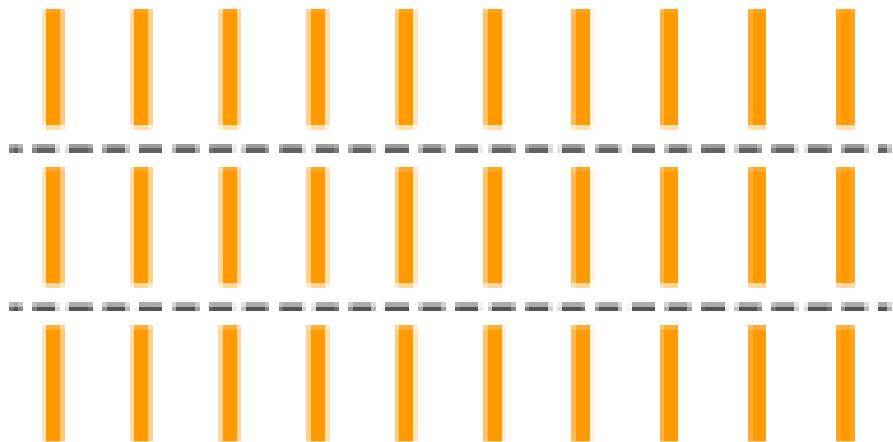
Análise do LSH

Análise do processo

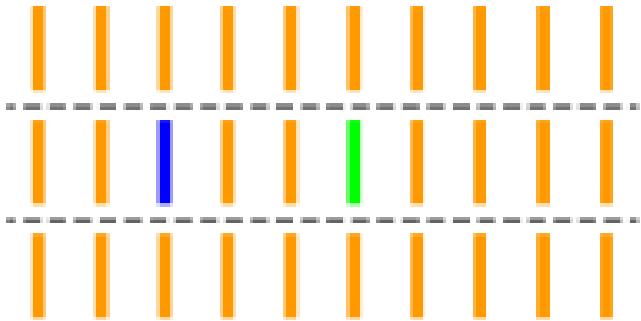
- Assumimos que:
 - existem **posições (buckets) suficientes** para que as colunas não sejam suscetíveis de mapeamento para a mesma posição a menos que sejam **idênticas** num banda específica
- Em consequência, **assumimos que “mesma posição” significa “idêntico nessa banda”**
- Esta simplificação é necessária **apenas para simplificar a análise**, não para a correção do algoritmo

b bandas, r linhas/banda

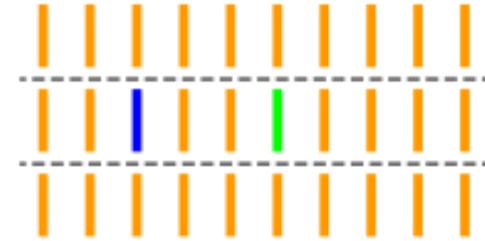
- É conveniente representar as bandas de forma abreviada
- A Figura refere-se a 10 documentos (10 colunas) e 3 bandas.



Análise (continuação)



- Consideremos os dois blocos marcados com coloração diferente na figura
- A probabilidade de todos os elementos do bloco a azul serem iguais aos elementos do bloco verde é J^r
 - J é a similaridade de Jaccard dos dois
 - resultante de as colunas C_1 e C_2 terem semelhança J
 - r é o número de linhas de uma banda



Probabilidades

- A probabilidade de que nem todos os elementos nos dois blocos sejam iguais é

$$1 - J^r$$

- Nem todos serem iguais em b bandas tem probabilidade

$$(1 - J^r)^b$$

- Probabilidade de pelo menos uma banda idêntica:

$$P = 1 - (1 - J^r)^b$$



Exemplo de aplicação

- Consideremos o seguinte caso:
- 100 000 documentos (\Rightarrow 100 000 colunas de M)
- Assinaturas de 100 inteiros (linhas)
 - Ocupam 40 Mb
- Com $b = 20$ bandas de $r = 5$ inteiros/banda
- **Objetivo:** encontrar pares de documentos que apresentem $s \geq 0.8$

Caso 1: $\text{sim}(C_1, C_2) = 0.8$

- $b = 20$ bandas de $r = 5$ inteiros/banda.
- Vejamos primeiro o que acontecerá a dois documentos com similaridade elevada (0.8)
- Como $\text{sim}(C_1, C_2) \geq s$ pretendemos que C_1, C_2 seja um par candidato
 - Pretendemos que haja pelo menos uma banda idêntica
- Probabilidade de C_1, C_2 idênticas numa banda específica:
$$J^r = (0.8)^5 = 0.329$$
- Probabilidade C_1, C_2 não serem semelhantes em todas as 20 bandas:
$$(1 - 0.329)^{20} = 0.00035$$
 - Cerca de 1/3000 dos pares de colunas semelhantes a 80% são falsos negativos
- Encontraríamos 99.97% de pares de documentos verdadeiramente semelhantes

Caso 1: $\text{sim}(C_1, C_2) = 0.8$

- $b = 20$ bandas de $r = 5$ inteiros/banda.
- Considerando agora documentos com baixa similaridade
 $\text{sim}(C_1, C_2) = 0.3$
 - Pretendemos que todas as bandas sejam diferentes

- Probabilidade de C_1, C_2 idênticas numa banda:

$$J^r = (0.3)^5 = 0.00243$$

- Probabilidade de termos C_1 e C_2 idênticos em pelo menos uma das 20 bandas:

$$1 - (1 - 0.00243)^{20} = 0.0474$$

- Aproximadamente 4.74% pares de documentos com similaridade de 0.3% acabam como pares candidatos, aparecendo como falsos positivos,
 - que terão de ser analisados mas não são de fato pares similares.

Otimização do processo

- A aplicação do LSH obriga a um **compromisso entre os falsos positivos e falsos negativos**
- Este compromisso pode ser efetuado através de:
 1. número de funções de dispersão
 - que resultam no número de linhas da matriz de assinaturas;
 2. número de bandas
 3. número de linhas por banda

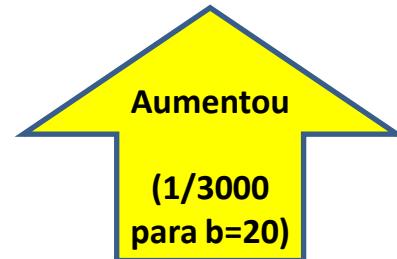
Exemplo - efeito de número de bandas

- Quais as alterações na probabilidade de falsos positivos e falsos negativos se utilizarmos apenas 15 bandas de 5 linhas, considerando os casos de $\text{sim}(C_1, C_2) = 0.8$ e $\text{sim}(C_1, C_2) = 0.3$?
- **Falsos positivos**
 - $P[C_1, C_2 \text{ idênticos numa faixa}]$: $(0.3)^5 = 0.00243$.
 - $P[C_1, C_2 \text{ idênticos em pelo menos 1 das 15 bandas}]$:
$$1 - (1 - 0.00243)^{15} = 0.0358$$
 - Aprox. 3.6% dos pares de documentos com similaridade 0.3 falsos positivos

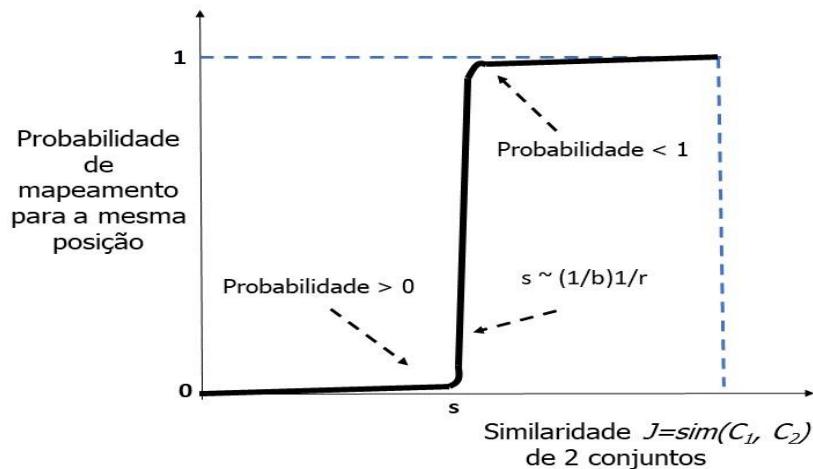
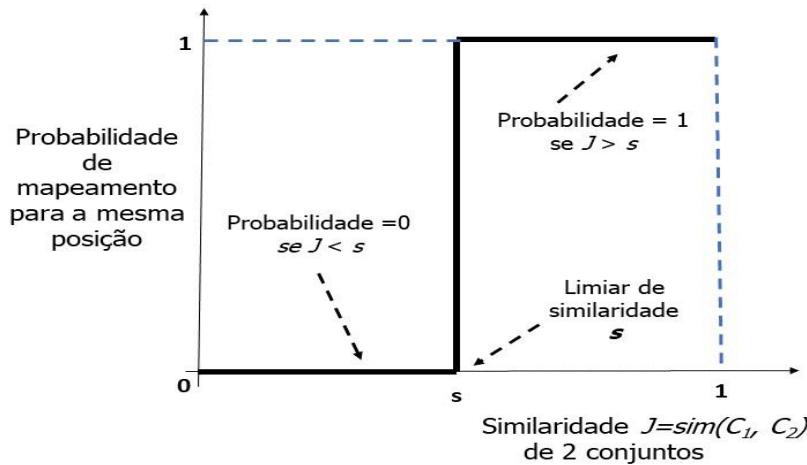
Diminui
(4.74%
para b=20)

Exemplo - efeito de número de bandas

- **Falsos negativos**
 - $P[C_1, C_2 \text{ idênticos numa faixa específica}] : (0.8)^5 = 0.328.$
 - $P[C_1, C_2 \text{ não semelhantes em todas as 15 bandas}] : (1 - 0.328)^{15} = 0.0026,$
 - cerca de 1/400 dos pares semelhantes a 80% são falsos negativos



Realidade versus situação ideal



- O que gostaríamos de ter
- Probabilidade nula para similaridades até ao limiar e probabilidade igual a 1 para pares com similaridade superior a esse limiar

- Desempenho real do processo LSH
- Temos probabilidades não nulas para valores de similaridade baixos
- Não temos valores iguais a 1 para similaridades elevadas

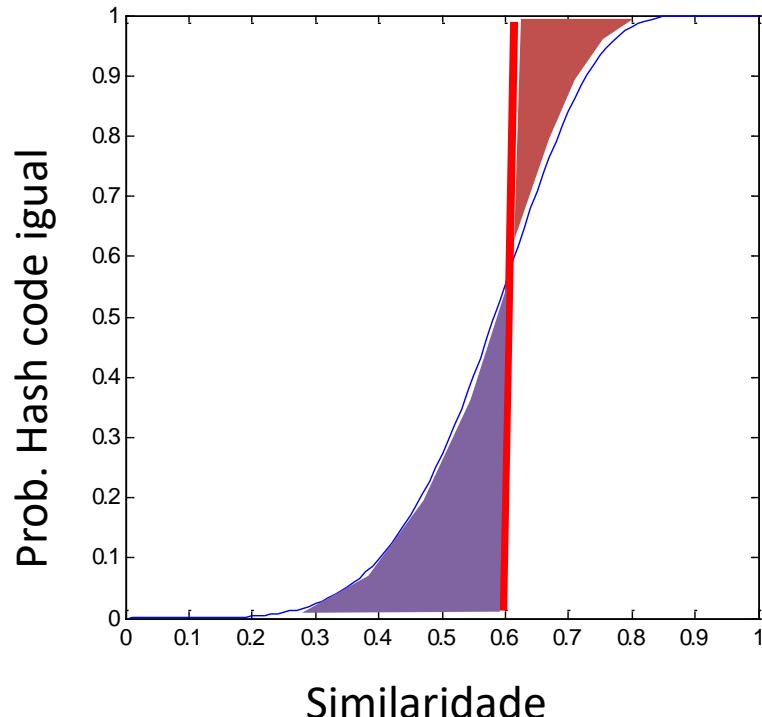
Exemplo

- $b = 20$ e $r = 5$
- Prob. de pelo menos uma banda idêntica

s	$1-(1-s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Valores para r e b

- **Seleccionar r e b para ter a melhor curva**
 - Ex: 50 funções de dispersão ($r=5$, $b=10$)



Área azul: Falsos Negativos

Área vermelha: Falsos Positivos

Obtenção dos parâmetros b e r

- b e r podem ser obtidos através da resolução de um **conjunto de equações**
 - definido em termos de verdadeiros e falsos positivos considerados adequados à aplicação
 - e/ou verdadeiros e falsos negativos

Exemplo de obtenção de b e r

- Pretendemos:
 1. Falsos positivos < 1% para todos os elementos com similaridade de Jaccard ≤ 0.6
 2. Verdadeiros positivos >99% para elementos com similaridade ≥ 0.9

Exemplo de obtenção de b e r

- Sistema de equações:

$$\begin{cases} 1 - (1 - 0.6^r)^b = 0.01 \\ 1 - (1 - 0.9^r)^b = 0.99 \end{cases}$$

- Como resolver?

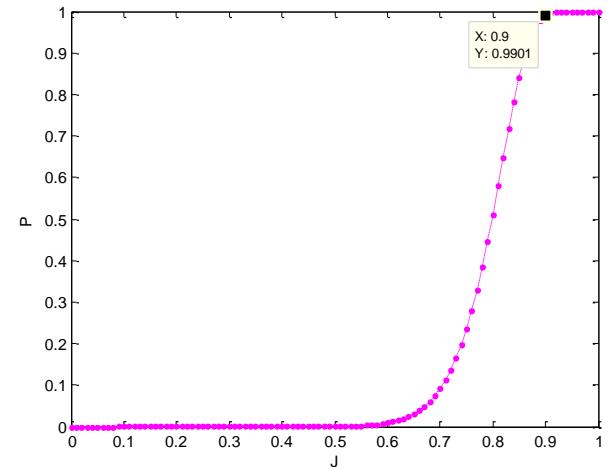
Resolução do sistema de equações

O sistema pode, por exemplo, ser resolvido em Matlab usando a Optimization Toolbox

```
fun = @probabilidade;  
  
x0 = [randi(10),randi(10)];  
x = fsolve(fun,x0);  
  
r=fix(round(x(1)))  
b=fix(round(x(2)))  
  
%% -----  
function F = probabilidade(x)  
  
r=x(1);  
b=x(2);  
  
F(1)= 1-(1-0.6^r)^b -0.01;  
F(2)= 1-(1-0.9^r)^b - 0.99;
```

Exemplo (continuação)

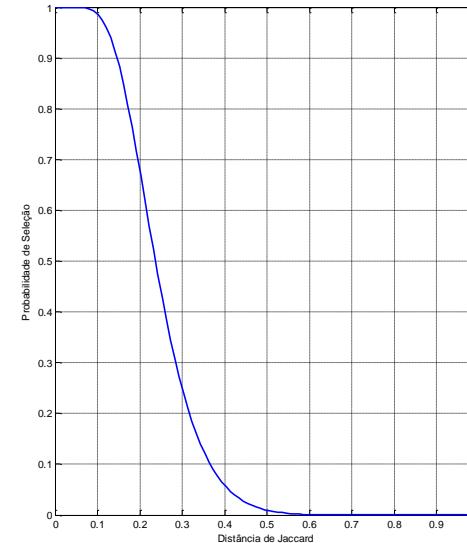
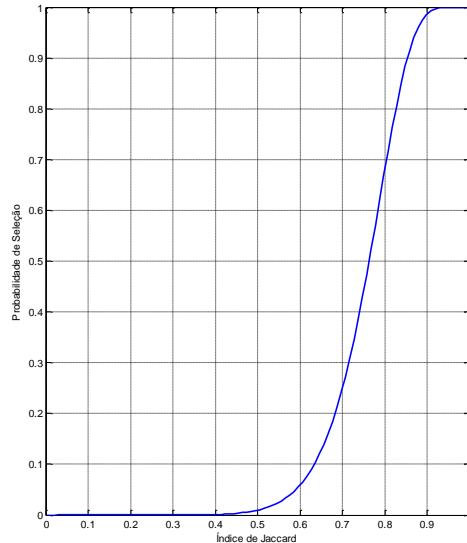
- Confirmemos os resultados...
- Curva $P(J) = 1 - (1 - J^r)^b$
 - $r=15$
 - $b=20$



- Probabilidade < 0.01 para similaridade de Jaccard <=0.6
- $1 - (1 - 0.6^{15})^{20} \approx 0.0095 < 0.01$
 - OK
- Probabilidade >0.99 para similaridade de Jaccard >=0.9
- $1 - (1 - 0.9^{15})^{20} \approx 0.9901 > 0.99$
 - OK

Aplicação ao MovieLens

- Processar a matriz Min-Hash
 - Já calculada
- Usemos, por exemplo:
 - $r=10$ $b=\text{NumHashFunctions} / r$



Comentários finais

- Afinar M , b , r para obter quase todos os pares com assinaturas similares
 - Mas eliminando a maior parte dos pares que não têm assinaturas similares
- Testar se os pares candidatos têm, de facto, assinaturas similares
- Opcional: Num outro passo, verificar se os pares candidatos remanescentes representam mesmo documentos similares

Recomendação

- Para informação adicional sobre este tópico recomenda-se a consulta de
Mining of Massive Datasets,
da autoria de Jure Leskovec, Anand Rajaraman e Jeff Ullman

Note to other teachers and users of these slides: We would be delighted if you found this material useful in giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: <http://www.mmds.org>

Parte dos slides adaptados de: Finding Similar Items: Locality Sensitive Hashing

Mining of Massive Datasets

Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University

<http://www.mmds.org>

