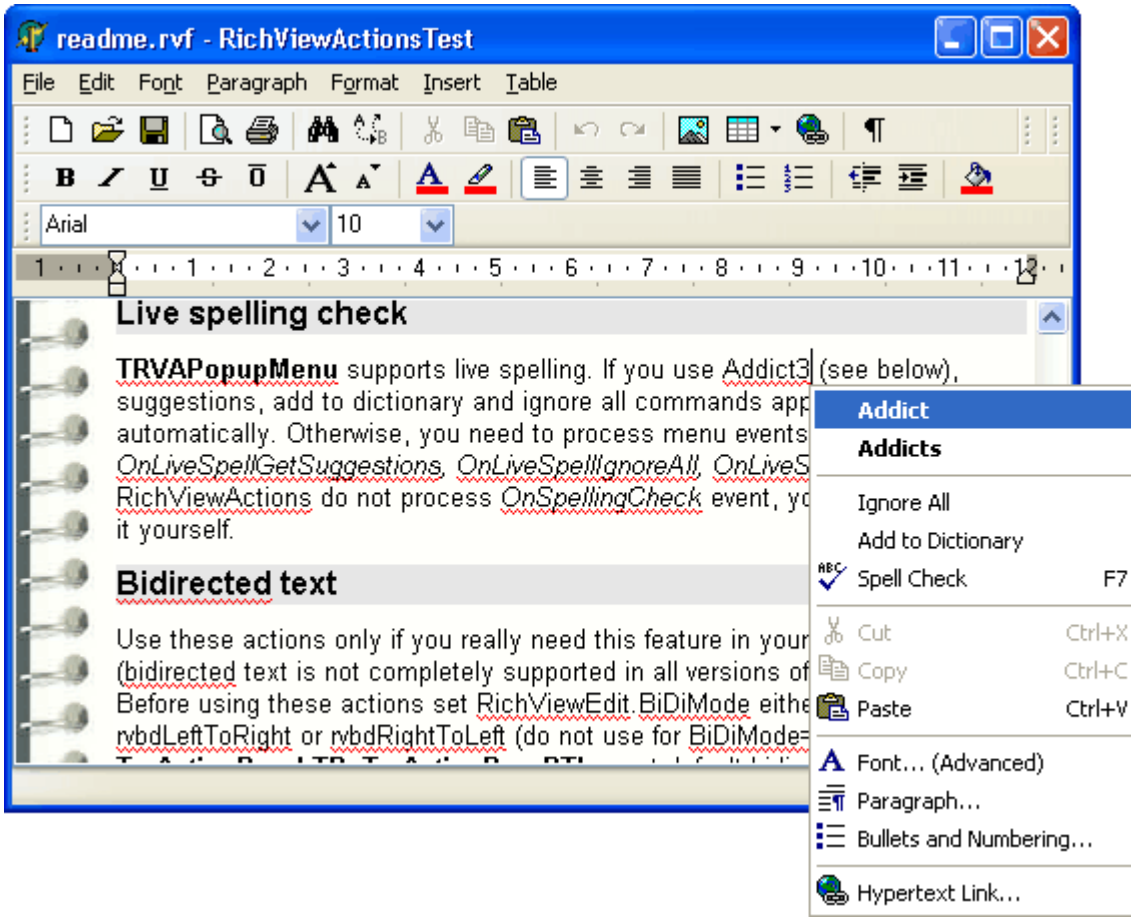
MPEI

Solução Probabilística
do
Problema da Pertença a um Conjunto

Problema 1 - Verificação ortográfica

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



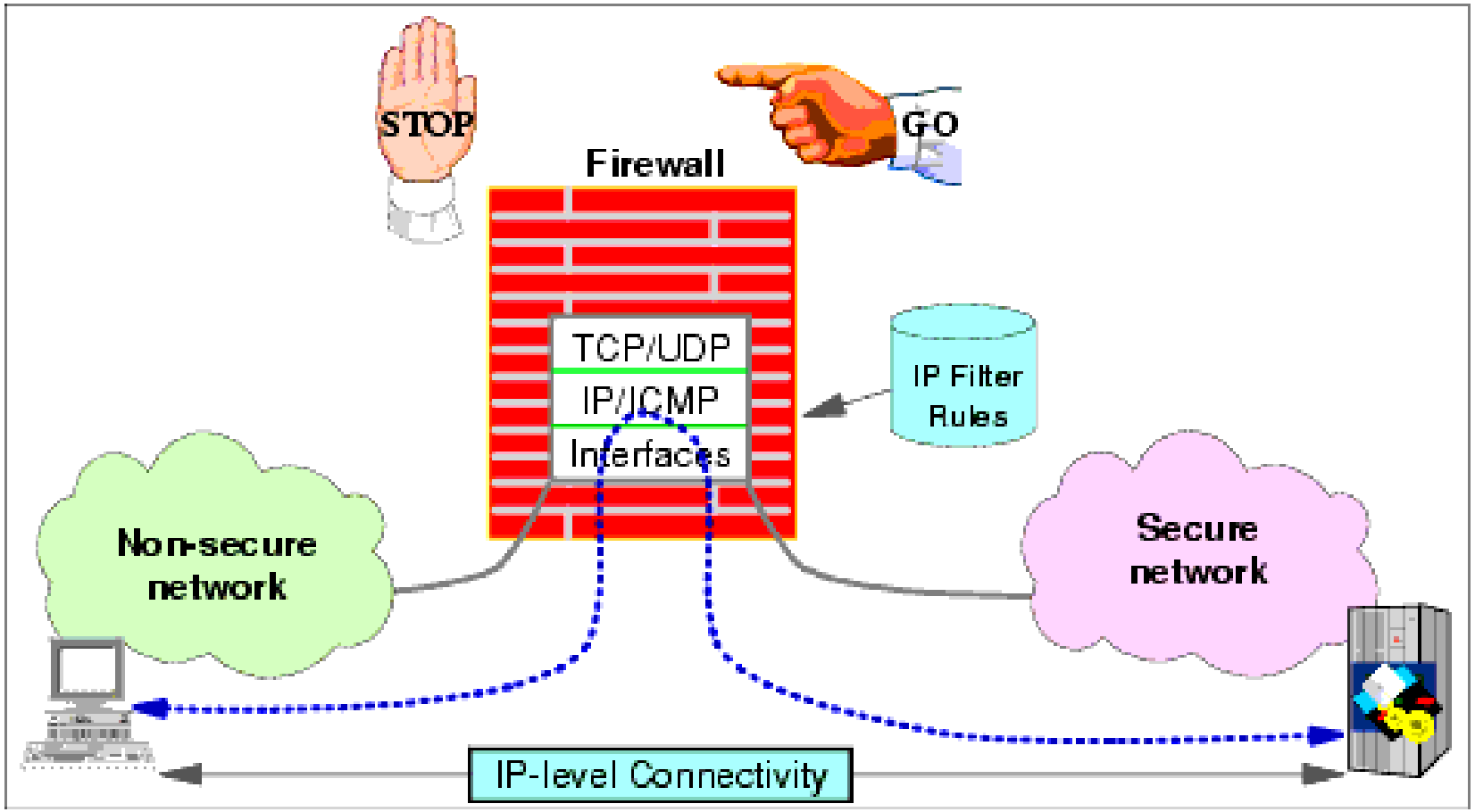
Eye have a spelling chequer,
It came with my Pea Sea.
It plane lee marks four my revue
Miss Steaks I can knot sea.

Eye strike the quays and type a whirred
And weight four it two say
Weather eye am write oar wrong
It tells me straight a weigh.

Eye ran this poem threw it,
Your shore real glad two no.
Its vary polished in its weigh.
My chequer tolled me sew.

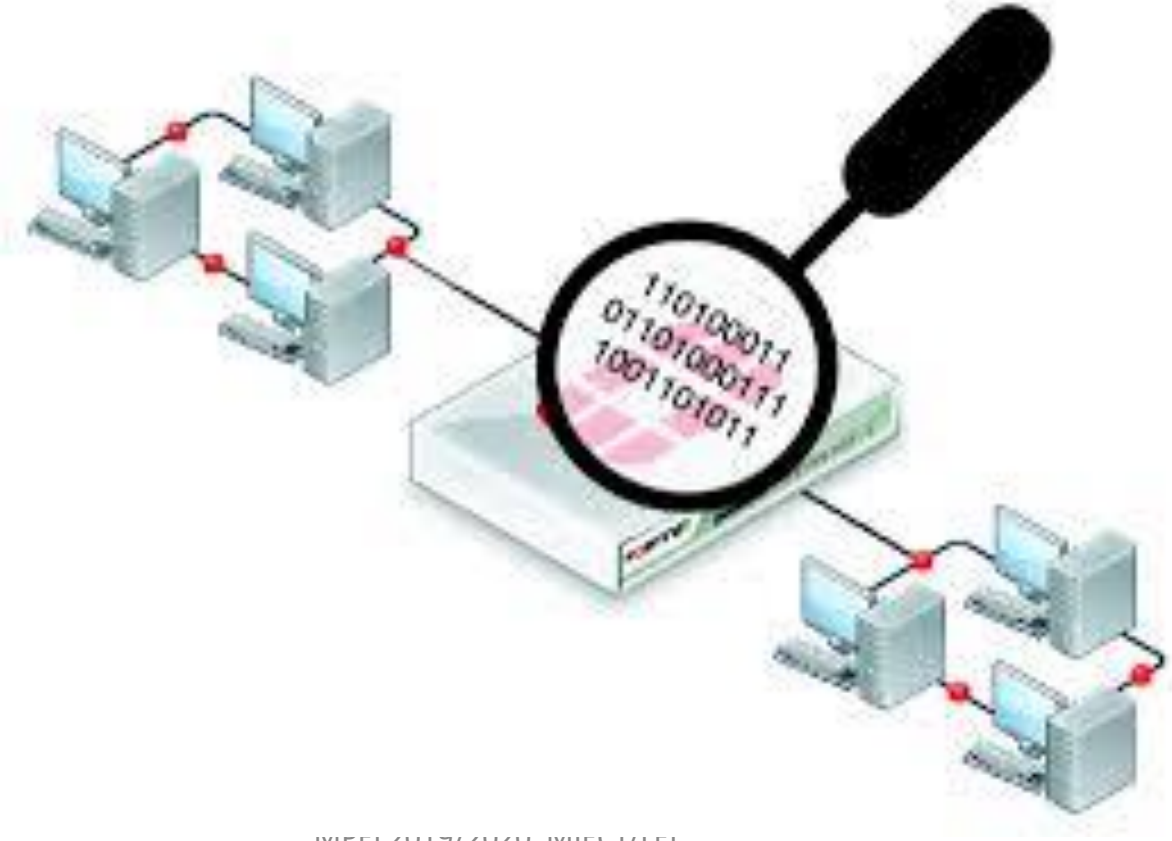
Uso cego de dicionários pode dar muito maus reultados

Problema 2 - Firewall



Problema 3

- Detetar sequências de bits (ex: Strings)



Generalizando …

- Na área da Informática temos muitos outros problemas similares
  - 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

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

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 a aos elementos

- Nestas situações podemos eliminar a parte de armazenamento dos elementos
  - guardando nele apenas informação de que existe

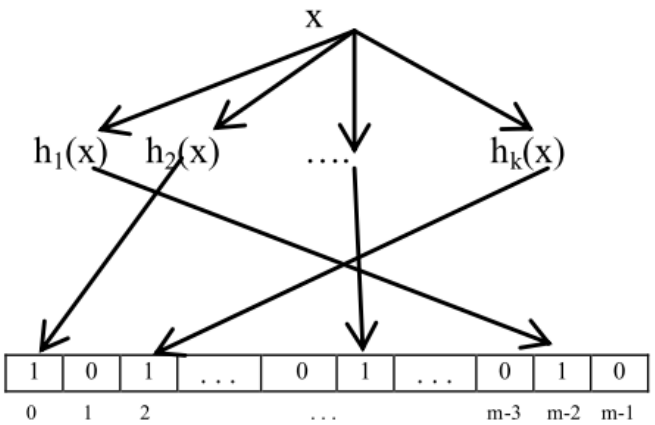
Filtros de Bloom

(Bloom Filters)

Os Filtros de Bloom são uma forma de usar funções de dispersão para determinar a pertença de um element a um conjunto

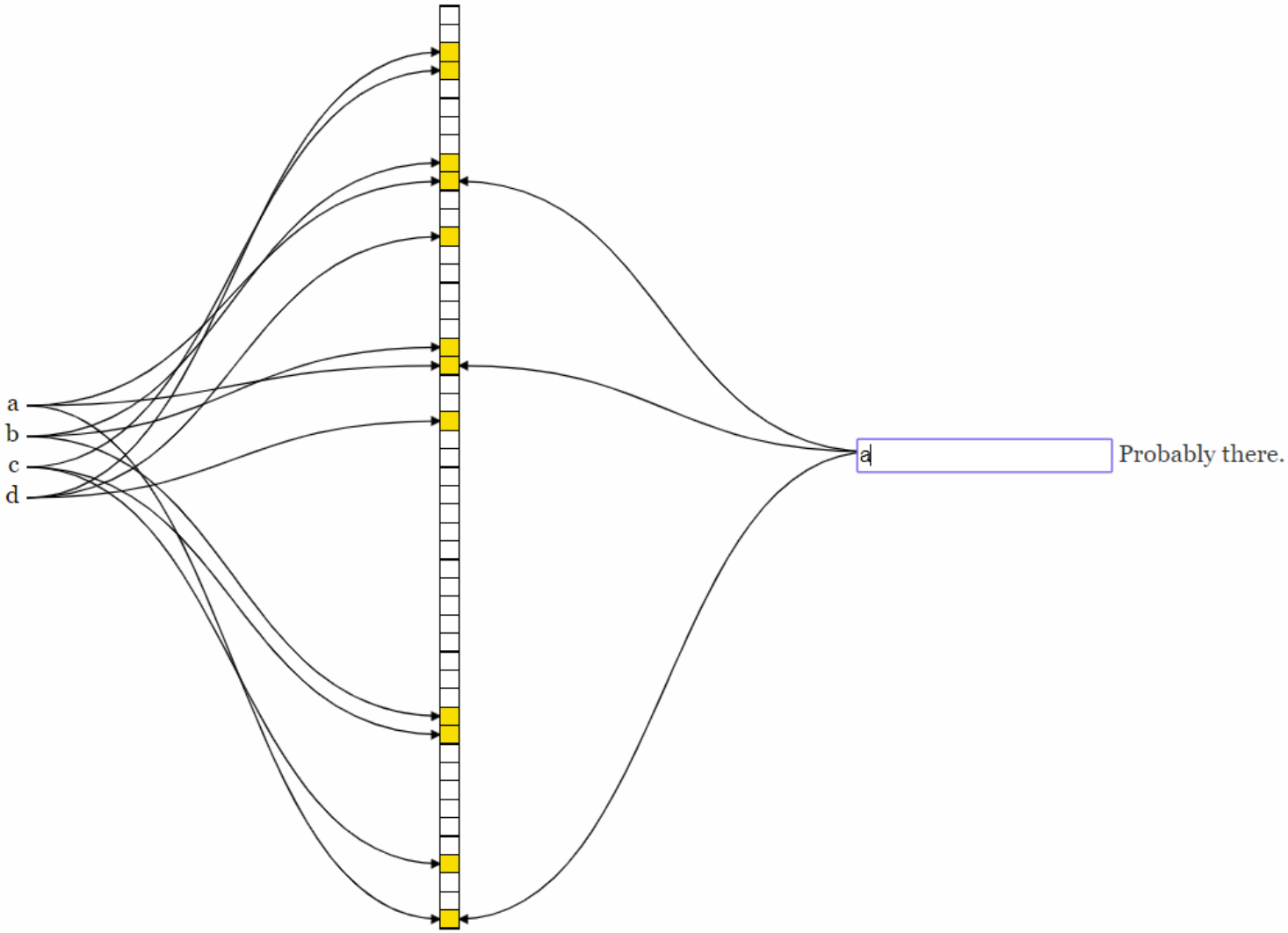
Filtros de Bloom

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



- A pertença ao conjunto é determinada através do conteúdo desse vetor nas posições devolvidas pelas mesmas funções de dispersão quando aplicadas a potenciais membros

Filtro de Bloom

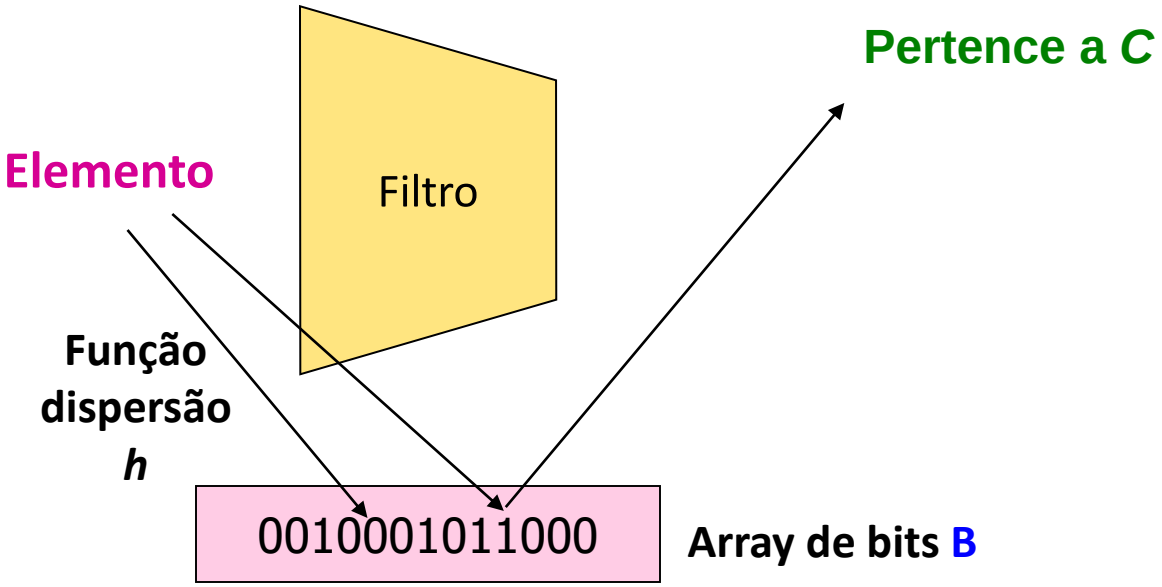


Filtros de Bloom

- 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
- Não incorporam qualquer tentativa de resolução de colisões

Filtro



Descartar o elemento

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

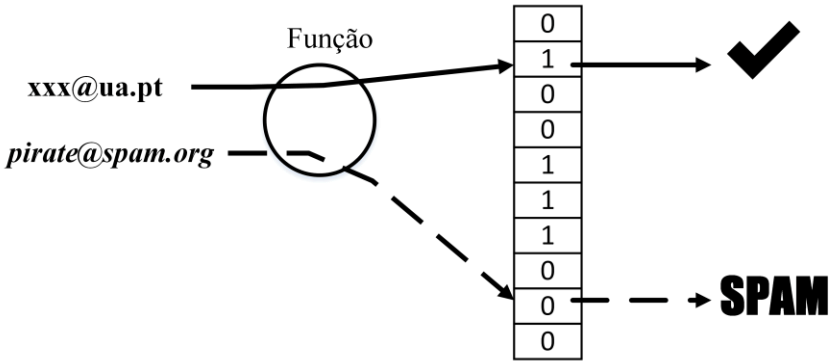
Exemplo – Deteção de SPAM

- 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 negativo**s.

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 **C**onjunto
    - com *m* elementos (membros) (#*C* = *m*)
  - *B* o vetor (filtro) de **B**loom
    - de dimensão *n* (#*B* = *n*)
  - *k* funções de dispersão independentes $h_1,\ldots,h_k$

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 1 todas as posições devolvidas pelas funções de dispersão
    - *Ou seja* B[h_i (*elemento*)] = 1

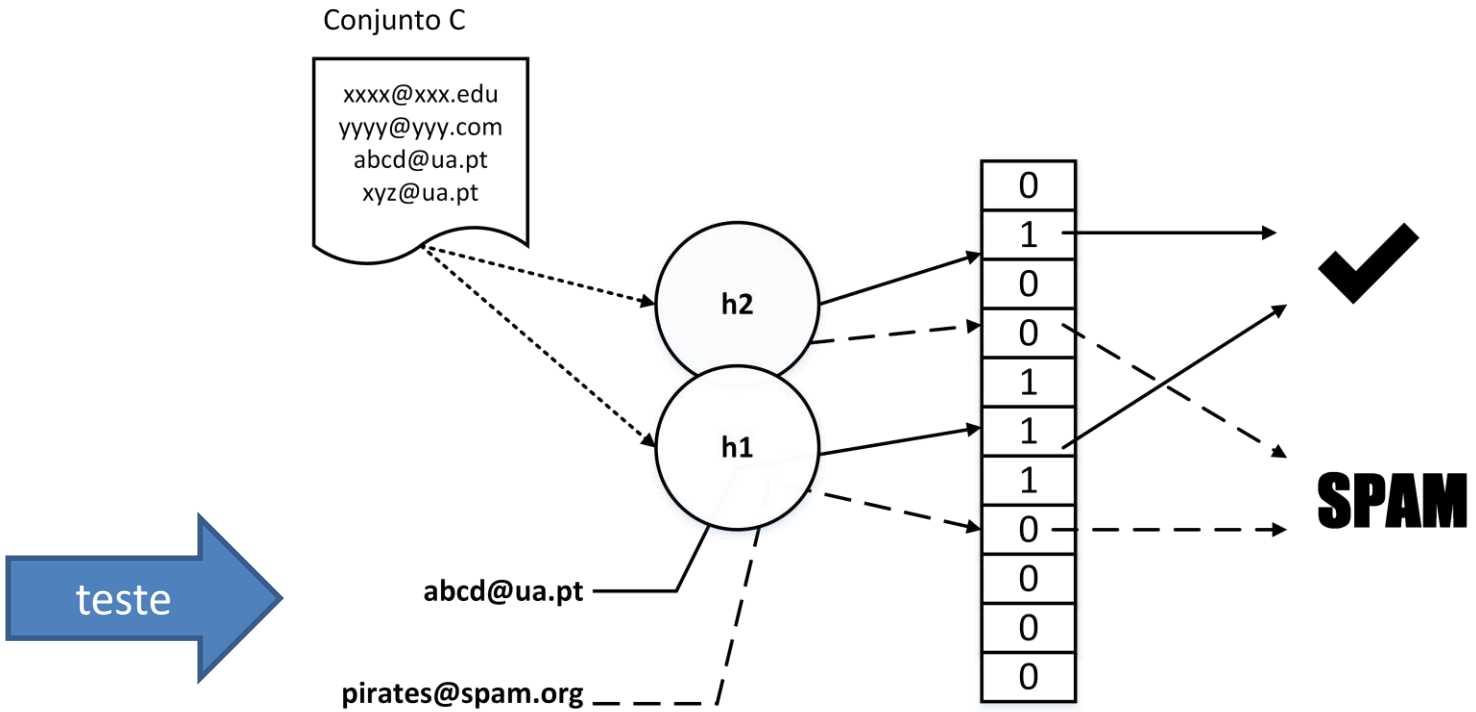
Utilização do Filtro de Bloom

- 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 $i = 1,...,k$ 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 pertença 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

- Consideremos 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 dipersão a **Tomate**:

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

- O filtro dafigura 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
  - teste de pertença de um elemento ao conjunto
  - e adição de elementos
- 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<=k)
        h é o hash code devolvido por hi (elemento)
until (   (i==m) | (B [h] == 0)   )

if i==m  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 originário
  - 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 string 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/

Questões?

- Como obter n ?

  (números de posições do filtro)

- 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 ?	Resulto 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 pelo menos um alvo seja 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

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

  - 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 ser zero para as *k* funções de dispersão
- Se assumirmos que são independentes, será:

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

Probablilidade após inserir m elementos

- Após a inserção de m elemento, 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 pelo menos por 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

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

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

- … pode ser aproximada por:

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

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

Efeito de k na p_{fp}

- Exemplo:
  $m = 10^9$ (mil milhões)
  $n = 8 \times 10^9$ (8 mil milhões)

m/n= 1/8

- k=1

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

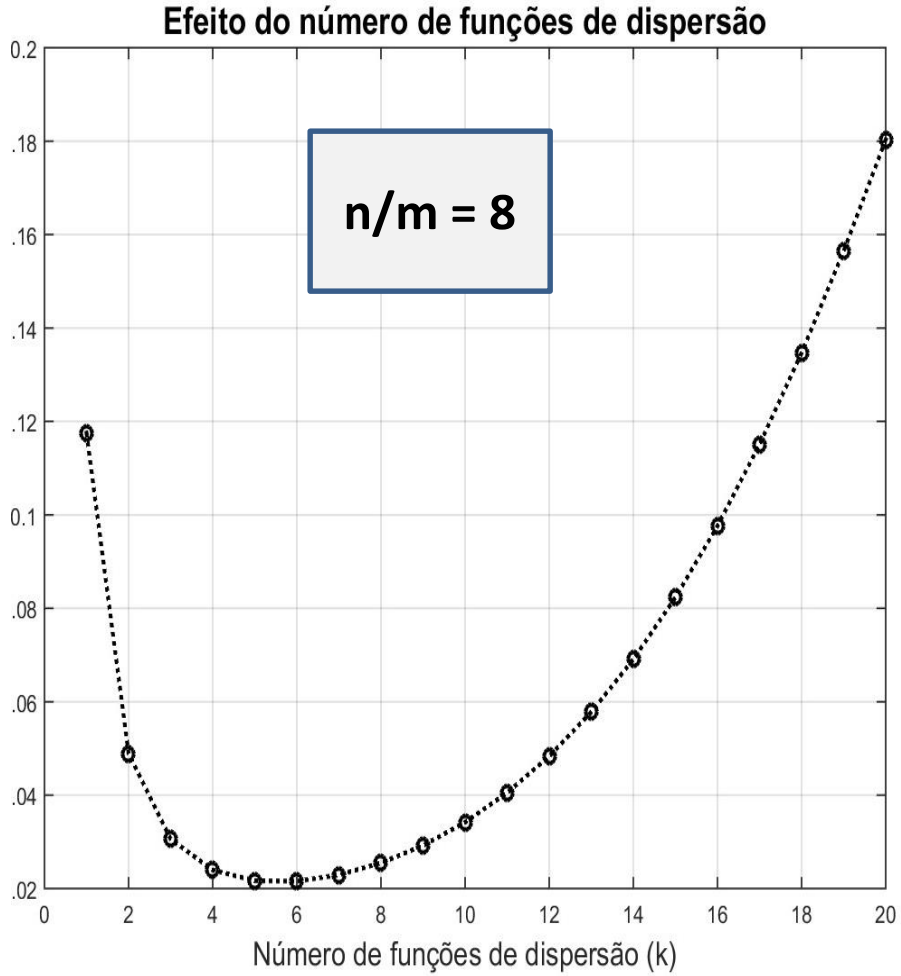
- k=2

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

- Probabilidade de erro diminui com aumento de k

- O que acontece se formos aumentando k ?

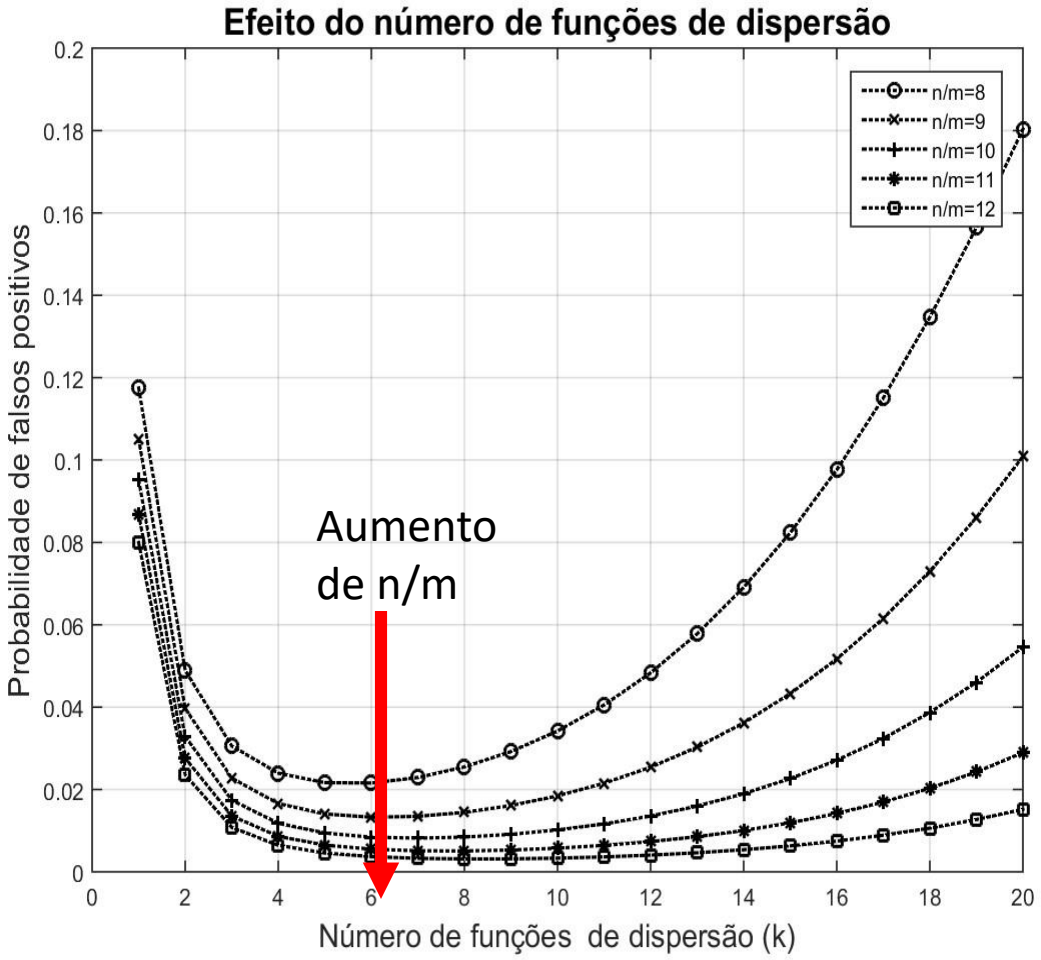
Efeito de k na prob. Falsos Positivos



- 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 ?

- E 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

$$\left(1-a^k\right)\ln\left(1-a^k\right)-a^k\ln\left(a^k\right)=0$$

- 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*:

$$k_{ótimo} = \frac{ln(1/2)}{ln(a)}$$

- Substituindo o valor de a temos

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

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

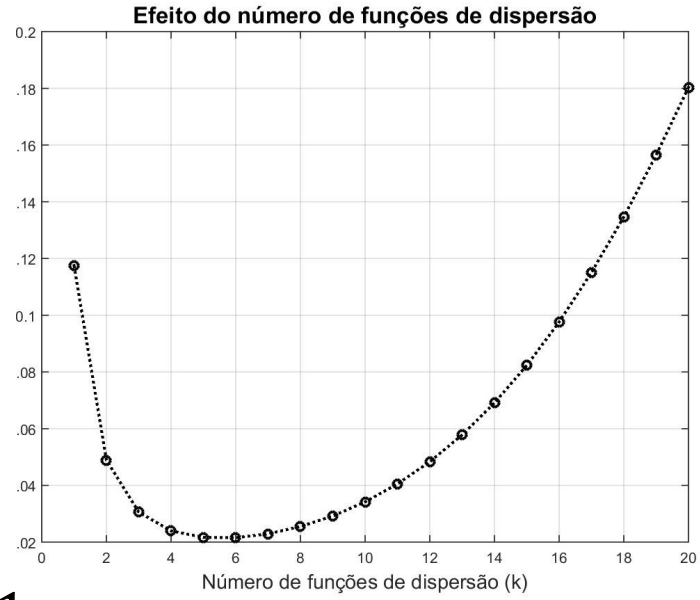
$$k_{ótimo} \approx \frac{\boldsymbol{n \ \ ln(2)}}{\boldsymbol{m}} = \frac{0.693\, n}{m}$$

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

Exemplo – k ótimo

- m= 10^{12}
- n=8×10^{12}

- k ótimo ?



- $k_{ótimo} \approx \frac{n \ln(2)}{m} = \frac{0.693 \times 8 \times 10^{12}}{10^{12}} = 5.54$
  - 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

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

  - *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 $k_{ótimo}$, a equação $\left(1 - a^k\right)^k$ resultaria em:

$$p_{ótima} = \left(1 - \frac{1}{2}\right)^{k_{ótimo}}$$
$$= \left(\frac{1}{2}\right)^{k_{ótimo}}$$
$$= 2^{-k_{ótimo}}$$

- 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 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 paralelizadas

Filtros de Bloom - Limitações

- Como a tabela não pode ser expandida, o máximo 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
  - E do espaço alocado para armazenar o vetor

Comentários finais

- Os filtros de Bloom d**evem ser considerados para programas em que um teste de pertença imperfeito pode ser aplicado** a um conjunto de dados (muito) grande

- A **grande vantagem** de um filtro de Bloom relativamente a uma única função de dispersão 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

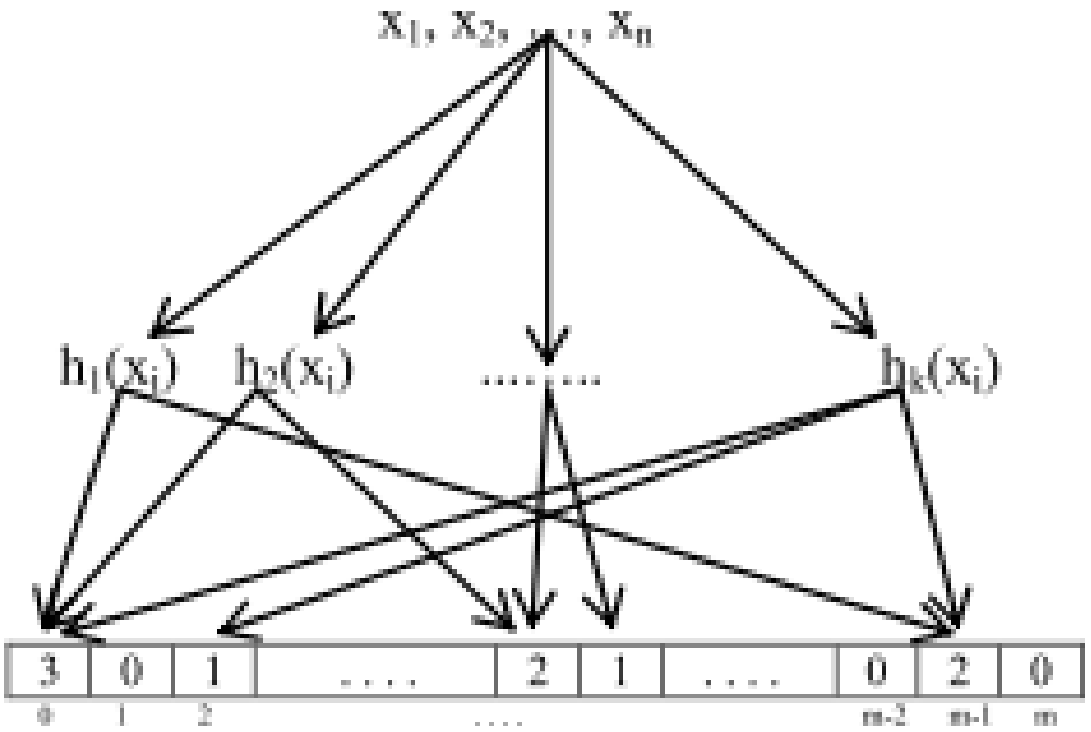
- As posições do vetor são estendidas de um único bit para um contador de *b* bits

  [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. Detemina-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 considerer como pertencendo ao conjunto

Problemas …

- *Overflow* do contador
  - Quando a contagem chega a $2^b - 1$
  - Tipicamente as implementações param de contar
    - Preferível a recomçar em 0
    - Mas introduz FN

- 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/

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