Problema

Em termos gerais o problema pode ser colocado desta forma:

Dado um elemento 'e', e um conjunto C, e pertence a C ?

A resolução deste problema para conjuntos pequenos é facial com recurso a vários algoritmos e estruturas como por exemplo as **hash tables**.

No entanto para conjuntos de maior dimensão não é assim tão simples sendo a razão o um dos recursos mais escassos na computação, memória.

Muitas vezes pode não haver memória suficiente para armazenar todos os elementos de **C**

Sendo portanto uma solução probabilística, com a crescente relevancia do **Big data**

Exemplo 1

Um destes problemas com que convivemos todos os dias é por exemplo a verificação ortográfica que:

- Utiliza dicionários com regras
- E têm a ortografia validada pela presença

Exemplo 2

Detetar sequências de strings num caixa de correio para filtrar o spam e executar regras

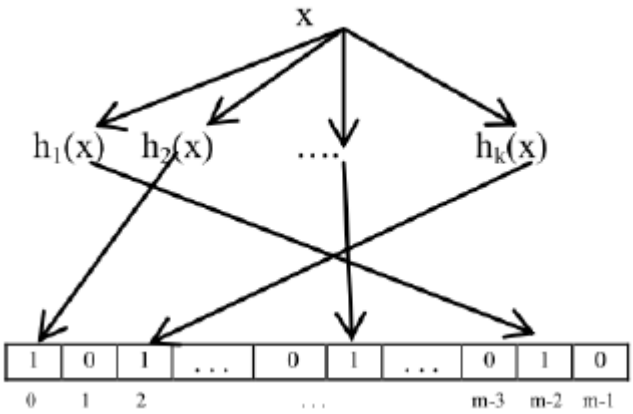
Ideia base

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

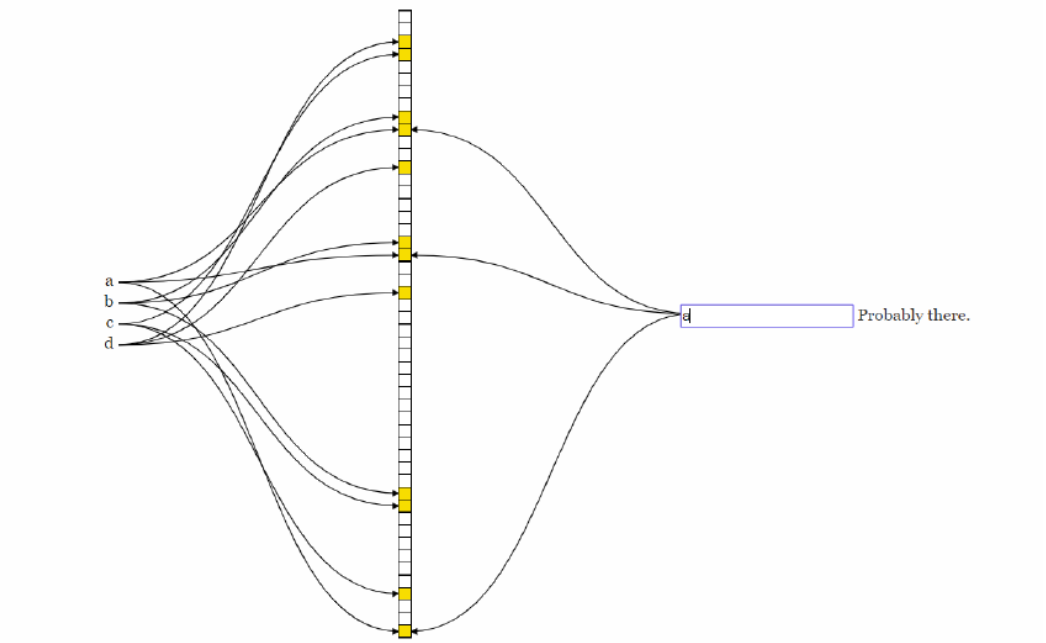
Nestas situações **podemos eleminar parte do amazernamento de dados** guardando apenas neles a informação que neles existe

Filtros Bloom

Os filtros de Bloom usam **hash functions** para calcular um vetor que é representativo de um conjunto



A pertença ao conjunto é testada através da comparação dos resultados da aplicação das mesmas **hash functions** aos potenciais membros com o conteúdo desse vetor

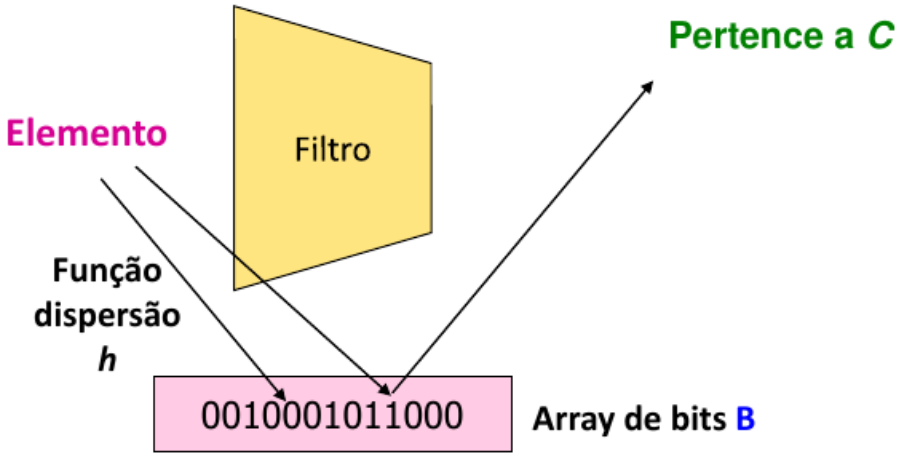


Na sua forma mais simples o vetor é composto de n posições

- Cada uma de apenas um 1 bit

O bit corresponde a um elemento é apenas colocado a 1 se a hash function mapear nessa posição algum dos elementos do conjunto

São rápidos, de complexidade temporal constante e não incorporam qualquer tentativa de resolução de colisões



Deteção de spam

Conhecem-se 10^9 de **endereços de email de confiança** que vão constituir o nosso conjunto **C**

Se algum dos emails recebidos não vierem desses emails serão movidos para o spam

Uma solução consiste em:

1. Criar vetor de n bits **B** bastante grande incializado todo a zeros
2. Utilizar uma hash function para mapear cada endereço de emai numa posição desse valor
3. Colocar a 1 os bits correspondentes à aplicação referida da hash function a toda a lista de email bons
4. Aplicar a hash function a cada email que se pretende verificar se calhar nos índices a 0 é **spam**

Ausencia de falsos positivos

Se um endereço que verificamos pertence a C então certamente que ele vai ser mapeado pela hash function para um índice do array que contêm o valor a 1

Generalização

A solução verificada anteriormente pode ser generalizada pela utilização de um conjunto de funções de dispersão para redundância no caso de falsos positivos

- Temos **C** como o conjunto, de dimensão **m**
- **B** o vetor de Bloom, de dimensão **n**
- **k** hash functions idependentes h1,h2,h3....hk

Inicialização de um filtro Bloom

1. Inicializar todas as posições de **B** a 0
2. Aplicar k hash funtions
    - B[hi (elemento)] = 1

Utilização do Filtro de Bloom

Para testar um valor x:

Aplicar as j funções de dispersão e analisar o conteúdo das posições resultantes se **B[hi (x)] == 1 para todos os valores de i = 1,...,k** então x provavelmente pertence a C

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.

Parâmetros do Filtro de Bloom

m: Dimensão do filtro

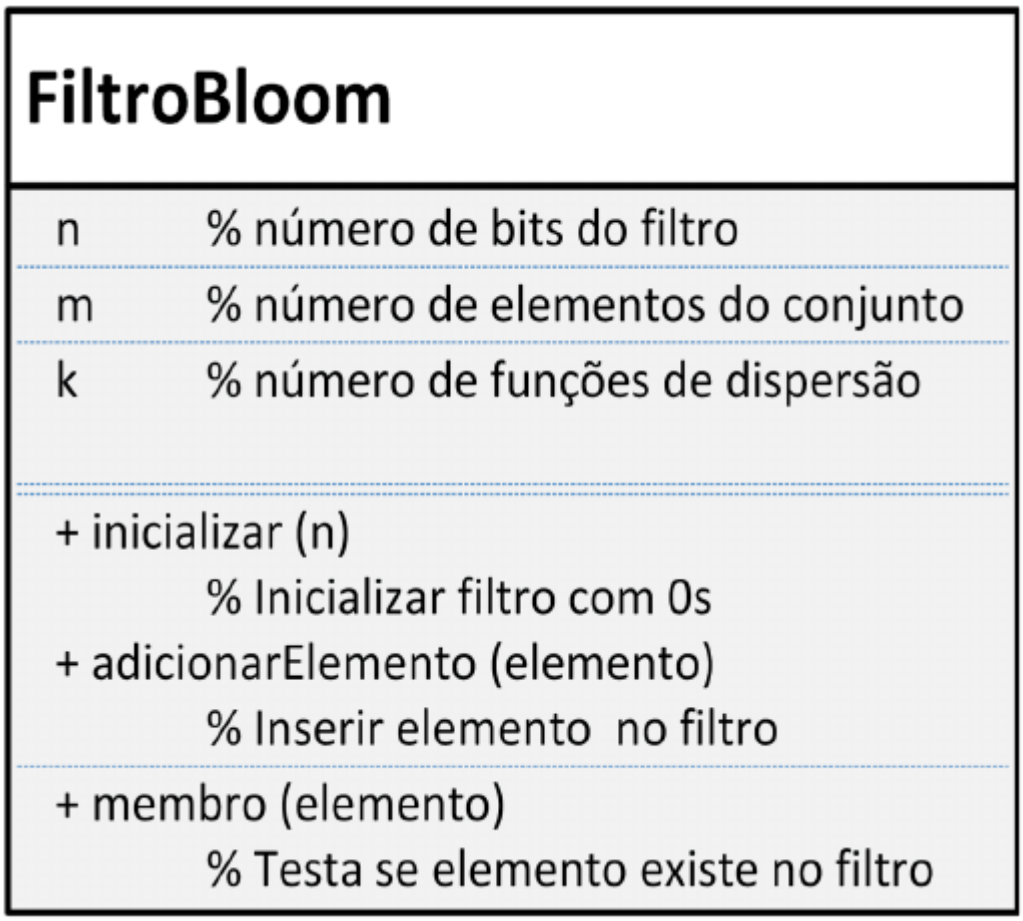
n: Número de posições ou células do filtro

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

- Inicialização
- Adição de elementos
- Teste de pertença a conjunto



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)

Aplicações

- Spell checkers
- Redes de dados
- Segurança e privacidade

Como obter os parâmetros para um filtro de Bloom

Nas aulas práticas já fizemos calculamos algo parecido se atirarmos **m** dardos para **n** alvos qual a probabilidade de acertarmos

No caso dos filtros de Bloom

- Os alvos são os vários bits do filtro
- Os dardos são os valores assumidos pela hash function

Relação n,k e m

Temos que descobrir a relação entre entres 3 termos de modo a obter um **filtro bloom** de com uma taxa de falsos positivos mínima

Probabilidade de falsos positivos

Para termos um falso positivo temos de acertar **k** posições determinadas pelas **hash functions** a 1

Probabilidade para 1 bit

Inicialmente todos os bits estão a zero, sendo **bi** o bit da posição **i** a probabilidade de a primeira **hash function**, assumindo que a **hash funtion** seleciona cada umas das posições com igual probabilidade é:

$$P[b_i = 1] = \frac{1}{n}$$
$$logo$$
$$P[b_i = 0] = 1 - \frac{1}{n}$$

Probabilidade para k hash functions

Se assumirmos que os resultados das hash functions são independentes podemos assumir o seguinte:

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

Probabilidade para vários elementos

Após a inserção de **m** elementos, continuando com a independência temos que:

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

Substituindo na expressão

$$(1 - \frac{1}{n})^m = a$$

Obtemos:

$$P[b_i = 0] = a^k$$
$$P[b_i = 1] = 1 - a^K$$

Temos um falso positivo quando temos **k** bits iguais a 1 para um elemento não pertencente a **C**.

Pelo que a probabilidade de um falso positivo depois da inserção de **m** elementos será

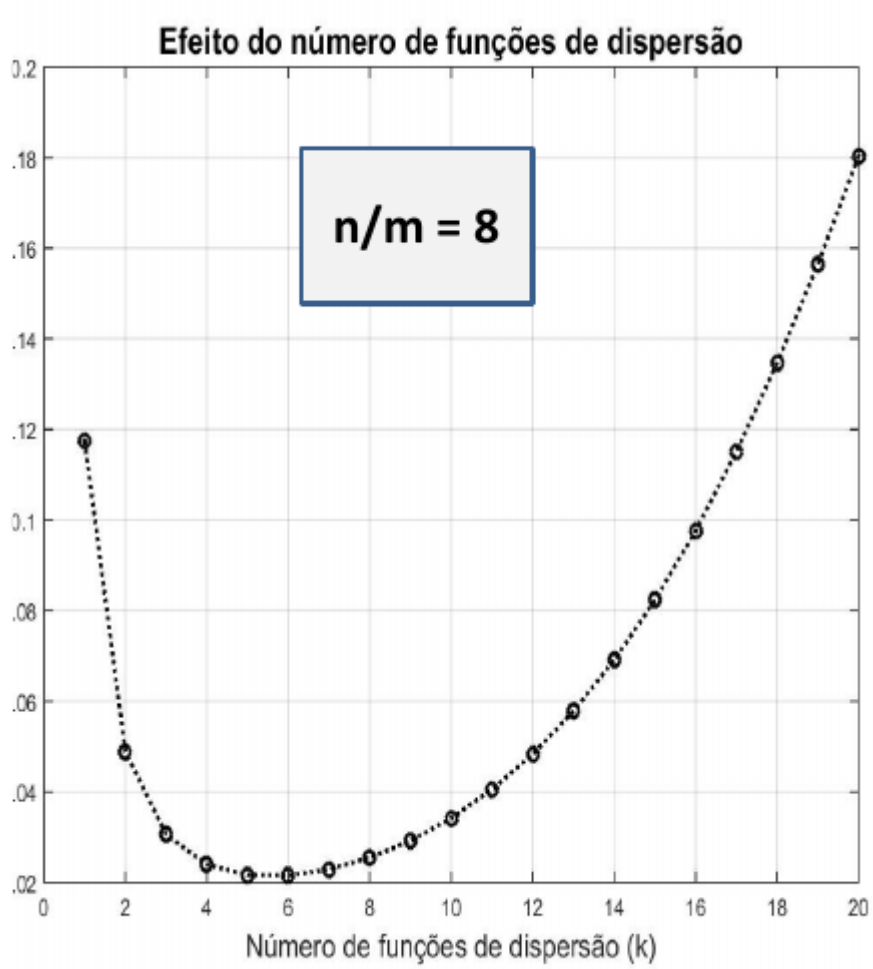
$$p_{fp} = [1 - (1 - \frac{1}{n})^{km}]^k = (1 - a^k)^k$$

Calculando o limite

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

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

Podemos ver a relação entre **k** e numero de erros neste gráfico onde com **m** e **n** fomos aumentando o valor de **k**



Depois de um certo **k** começamos a ver um crescimento no erro de novo pelo que o **k ótimo** será o zeros da função derivada

Portanto resolvendo a seguinte equação

$$(1 - a^k) * ln(1 - a^k) - a^k * ln(a^K) = 0$$

$$\dots$$

$$a^k = 1/2$$

podemos calcular agora o **k** a partir da expressão

$$k_{ótimo} = \frac{1/2}{ln(a)}$$

$$substituindo\ 'a'$$

$$k_{ótimo} = \frac{0.693n}{m}$$

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
- e assumindo que o valor de k ótimo é adotado

Sumário

Aspectos positivos

- Os filtros de Bloom garantem não existência de falsos negativos
- Usam uma quantidade de memória limitada
- Adequados para implementação de hardware

Limitações

- Numero máximo de elementos para armazenar no filtro têm de ser previamente conhecido
- Assim que se excede a capacidade para a qual foi projectado, os falsos positivos aumentam rapidamente ao serem inseridos mais elementos

Compromissos

- Aumento do número de funções de dispersão (até ao **k** ótimo)
- Aumento do espaço alocado para armazenar o vetor

Características

- Rapidez
- Capacidade de processar um enorme conjunto de dados
- Desadequado para conjuntos pequenos

Filtros de contagem

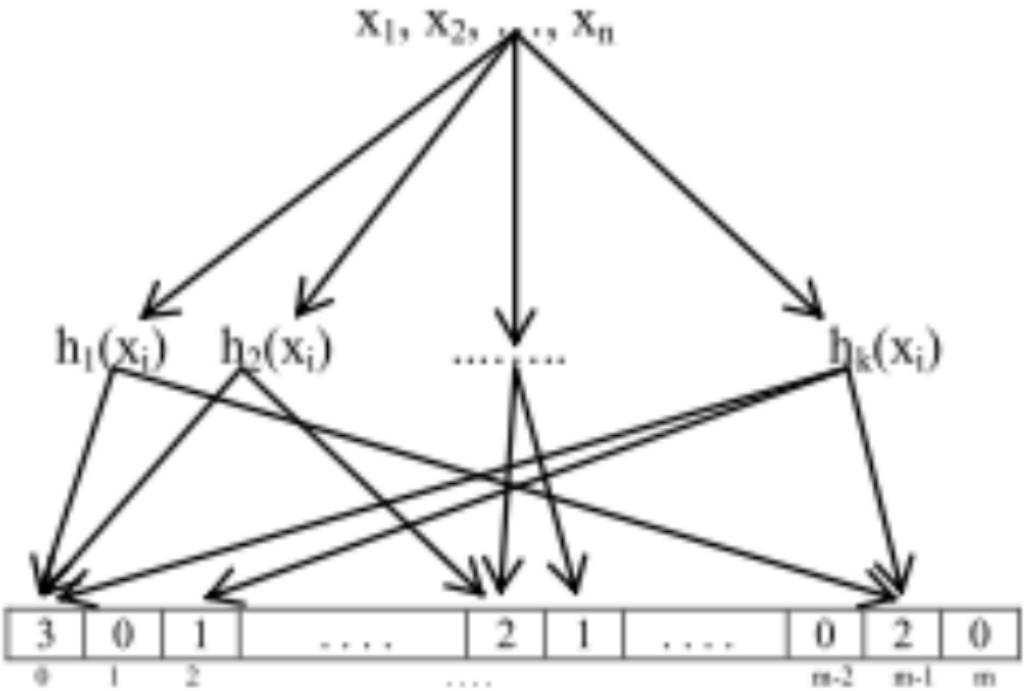
Um filtro Bloom básico representa um conjunto, mas:

- Não permite a consulta da multiplicidade
- Nem suporta a remoção de elementos

Este filtro vêm respostas ás limitações do filtro de bloom

As posições agora em vez de serem apenas um **bit** são estendidas para um contador de bits

Na **inserção** de um elemento o contador é **incrementado** e na **remoção** é **decrementado**



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

Bastam ligeiras alterações a:

- adicionar() : passa a incrementar
- membro() : requer que todos sejam não nulos
- A criação da função adicional **contagem()**

contagem()

Para obter a contagem (multiplicidade) associada a um elemento de um 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

Problemas

- Overflow do contador
- 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