Capítulo 5

Algoritmos de Dados Massivos

5.1 Introdução

No estudo clássico da complexidade de algoritmos, um algoritmo é dito *eficiente* se sua complexidade de tempo é $O(n^{c'})$, onde n representa o tamanho da entrada e c' é uma constante não-negativa. Como cada célula de memória alocada por um algoritmo é normalmente inicializada, a complexidade de tempo de um algoritmo é maior ou igual àquela de espaço. Desta maneira, se um algoritmo é eficiente, sua complexidade de espaço é $O(n^c)$ para alguma constante não-negativa $c \leq c'$.

Tal definição de eficiência, na prática, é insuficiente, pois ela permite valores de c, c' arbitrariamente grandes. Um algoritmo com complexidade de tempo de $\Theta(n^{100})$ é eficiente pela definição, mas não encontra emprego prático. Para uma aplicação real, espera-se que os valores de c, c' sejam pequenos, não excedendo algumas unidades. Neste capítulo, estudaremos algoritmos para os quais o tamanho da entrada n é muito grande, além dos limites considerados usuais, de modo que as restrições sobre os valores permitidos de c, c' sejam ainda mais severas. A principal destas restrições será que os dados de entrada não poderão ser carregados na memória auxiliar. Como consequência, a leitura da entrada poderá ser feita apenas uma única vez, de forma sequencial. Tais algoritmos serão chamados de *algoritmos de dados massivos*.

Estas restrições encontram-se presentes no paradigma de fluxos de dados, que é formalmente descrito na próxima seção.

5.2 O Paradigma de Fluxo de Dados

Os algoritmos deste capítulo adotam um paradigma chamado de *fluxo de dados*. Formalmente, pressupõe-se que os dados de entrada são lidos, de uma fonte, um a um, até que não haja novos dados a serem lidos. A sequência de dados $S = a_1, a_2, \ldots, a_n$ produzida por tal fonte é chamada de *fluxo*. Cada a_i consiste de uma cadeia binária de, no máximo, ℓ bits. Supõe-se que o valor de ℓ seja conhecido antes da leitura dos dados, enquanto o valor de n não necessariamente o seja. No término da leitura do fluxo, pressupõe-se a computação de uma saída que depende dos dados $S = a_1, a_2, \ldots, a_n$, onde a_n foi o último dado lido.

Como exemplo de fluxo, considere a geração, em tempo-real, da sequência de acessos a um serviço eletrônico popular, de modo que cada a_i seja a identificação do cliente associado ao i-ésimo acesso (o nome deste usuário, seu endereço IP, etc.). Um exemplo de computação, ao término da leitura do fluxo, é aquele de determinar o número distinto de usuários presentes em S. Note que, se um usuário acessou o serviço em dois momentos, então $a_i = a_j$ para algum $i \neq j$. Outro exemplo é determinar o usuário que mais acessou o serviço, dentre várias outras perguntas de interesse prático. Embora o momento de realizar a computação desejada seja imprevisível (ou seja, n será desconhecido ao algoritmo), a necessidade desta computação ao término da leitura é sabida de antemão. Sendo assim, os dados $a_1, a_2, \ldots$ devem ser consumidos por um algoritmo cuja finalidade é reportar o resultado tão logo esteja caracterizado o fim do fluxo de dados.

Note que o tamanho da entrada é $O(\ell n)$. Além disso, a variável n, daqui em diante, denota o número de dados, ao invés do tamanho da entrada como empregado na introdução do capítulo. Supõe-se que o valor ℓn seja grande o suficiente para tornar inviável a tentativa de um algoritmo armazenar todos os elementos de S simultaneamente em memória. No entanto, o algoritmo deve guardar alguma informação relevante a respeito de cada a_i de modo que, ao término da leitura de S, uma saída de interesse seja produzida (Figura 5.1). Considera-se que uma complexidade de espaço de $O(\ell^c \log^d n)$, para algum par de constantes pequenas não-negativas c, d, é viável na prática mesmo para valores grandes de n. Assim, esta será a complexidade esperada dos algoritmos de dados massivos, quando possível.

O Exemplo 1 ilustra um problema para o qual o algoritmo projetado sob o paradigma clássico é adequado também ao paradigma de fluxo de dados.

Exemplo 1. Dado um fluxo de dados $S = a_1, a_2, \ldots, a_n$ tal que cada $a_i \in \{0, 1, \ldots, 2^\ell - 1\}$, determinar $\sum_{i=1}^{n} a_i$.

Neste problema, a abordagem determinística natural é aquela de acumular a soma s dos elementos lidos do fluxo, conforme Algoritmo 5.1. A função *próximo*(S) empregada é uma função especial que retorna o próximo elemento do fluxo, caso haja

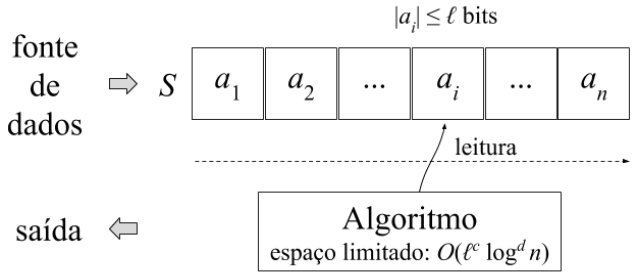

Figura 5.1: Paradigma para algoritmos que lidam com dados massivos.

algum, ou retorna um elemento especial $\varnothing$ no caso contrário, indicando o fim do fluxo. O algoritmo, tal como apresentado, também é de dados massivos, pois

$$s = \sum_{i=1}^{n} a_i \leq \sum_{i=1}^{n} 2^{\ell-1} = 2^{\ell-1} n = O(2^{\ell} n)$$

e, portanto, o espaço de memória $|s|$, necessário para armazenar s, é dado por

$$|s| = \log\Big(O(2^{\ell} n)\Big) = O(\ell + \log n)$$

o que de fato se encontra dentro do esperado para dados massivos. Adicionalmente, observe também que a hipótese de desconhecimento a priori do valor de n não acrescenta dificuldade na solução do problema.

Algoritmo 5.1 Soma de Elementos do Fluxo - Paradigma Clássico e de Dados Massivos

```
função SOMA-ELEMENTOS(S):
    s ← 0; a ← próximo(S)
    enquanto a ≠ ∅ :
        s ← s + a; a ← próximo(S)
    retornar s
```

Em contraste, o Exemplo 2 ilustra um problema, que é particularmente trivial sob o paradigma clássico, mas cuja solução no paradigma de dados massivos não é direta. Um algoritmo para este segundo problema, em conformidade com os requerimentos para ser um algoritmo de dados massivos, será apresentado na Seção 5.3.

Exemplo 2. Dado um fluxo de dados $S = a_1, a_2, \ldots, a_n$, onde n é desconhecido inicialmente, escolher um a_i de forma aleatória e uniforme.

Uma estratégia natural para solucionar o problema seria efetuar uma leitura até o final do fluxo, visando determinar o valor de n. Em seguida, pode-se realizar a escolha de um elemento com distribuição uniforme, ou seja, sortear um valor i no intervalo $1 \leq i \leq n$, e retornar o i-ésimo elemento. Para tanto, todos os elementos devem ser armazenados em memória durante a leitura do fluxo. Tal estratégia é a empregada na função Escolhe-Elemento do Algoritmo 5.2. Observe que a fila neste algoritmo pode ser implementada por uma lista encadeada, contornando o problema do desconhecimento prévio de n. A complexidade de espaço desta função é $O(\ell n)$ pois os elementos de fluxo precisam ser armazenados todos em memória.

Algoritmo 5.2 Escolha de um Elemento Aleatória e Uniformemente - Clássico

```
função Escolhe-Elemento(S):
    definir fila F
    n ← 0; a ← próximo(S)
    enquanto a ≠ ∅ :
        inserir a em F
        n ← n + 1; a ← próximo(S)
    escolher aleatoriamente inteiro i no intervalo 1 ≤ i ≤ n
    para j ← 1 até i :
        remover o próximo elemento de F, guardando-o em a
    retornar a
```

Há uma solução mais eficiente, de espaço $O(\ell + \log n)$, se n for previamente conhecido, conforme mostra a função Escolhe-Elemento-N do Algoritmo 5.3. Nela, pode-se efetuar imediatamente o sorteio da posição do elemento escolhido e desprezar o armazenamento de todos os elementos do fluxo exceto aquele de posição i.

Algoritmo 5.3 Escolha de um Elemento Aleatória e Uniformemente - Clássico

```
função Escolhe-Elemento-N(n, S):
    escolher aleatoriamente inteiro i no intervalo 1 ≤ i ≤ n
    para j ← 1 até i :
        a ← próximo(S)
    retornar a
```

Embora a complexidade de Escolhe-Elemento-N esteja dentro do esperado para um algoritmo de dados massivos, esta solução viola o fato de n ser previamente desconhecido. Por outro lado, Escolhe-Elemento não comete tal violação, mas não se qualifica para ser de dados massivos por armazenar todos os dados em

memória. Há um algoritmo que, sem pressupor o conhecimento de n, resolve o problema com complexidade de espaço também de $O(\ell + \log n)$. Este algoritmo será detalhado na Seção 5.3.

Nas próximas seções, apresentaremos problemas clássicos de dados massivos. Estes problemas têm caráter fundamental, com diversas aplicações.

5.3 Escolha Aleatória e Uniforme de Elemento

Nesta seção, retornamos à discussão iniciada no Exemplo 2 da Seção 5.1. O problema considerado é: dado um fluxo de dados $S = a_1, a_2, \ldots, a_n$, no qual n é desconhecido, escolher um elemento a_i aleatória e uniformemente. Vimos que o problema é trivial se n fosse conhecido, com espaço $O(\ell + \log n)$. Porém, deixamos em aberto uma solução de mesma complexidade quando previamente se desconhece n.

A solução encontra-se no Algoritmo 5.4. Para cada a_i lido, procede-se da seguinte maneira. Se $i = 1$, m é inicializado com o valor a_1. Para cada $i > 1$, troque o valor de m para a_i com probabilidade $1/i$ (e, portanto, mantenha o valor corrente de m com probabilidade $(i-1)/i$). Ao fim da leitura de S, o algoritmo reporta o valor de m como o elemento escolhido.

Algoritmo 5.4 Escolha de um Elemento Aleatória e Uniformemente - Dados Massivos

função ESCOLHE-ELEMENTO(S):
    $a \leftarrow$ *próximo*(S)
    $i \leftarrow 1$; $m \leftarrow a$
    $a \leftarrow$ *próximo*(S)
    enquanto $a \neq \varnothing$ **:**
        $i \leftarrow i + 1$
        escolher aleatoriamente real p no intervalo $0 \leq p \leq 1$
        se $p \leq 1/i$ **então**
            $m \leftarrow a$
        $a \leftarrow$ *próximo*(S)
    retornar m

A complexidade de espaço do algoritmo é $O(\ell + \log n)$. Resta mostrar que a estratégia acima de fato resolve o problema proposto.

(Colocar exemplo pequeno.)

(Colocar exemplo real.)

Teorema 5.1. *Seja m o elemento retornado pelo Algoritmo 5.4. Para todo $1 \leq i \leq n$, a probabilidade que o valor de m seja escolhido como aquele de a_i é de $1/n$.*

Demonstração. Mostraremos por indução em n. Se $n = 1$, é trivial verificar que o algoritmo reporta a_1, que é o elemento que deve ser sorteado de fato. Se $n > 1$, assuma que a estratégia acima esteja correta para todo fluxo de cardinalidade menor que n. Seja S' o fluxo $a_1, a_2, \ldots, a_{n-1}$. Uma execução do algoritmo até a iteração $i = n - 1$ no fluxo S pode ser vista como uma execução completa do algoritmo no fluxo S'. Sendo assim, por hipótese de indução, ao fim da iteração na qual $i = n - 1$ sob fluxo S, vale que m escolheu qualquer dos elementos de S' com probabilidade $1/(n-1)$. A probabilidade deste elemento ser mantido na iteração $i = n$ é de $\frac{n-1}{n}$. Sendo assim, a probabilidade de que um elemento dentre $a_1, a_2, \ldots, a_{n-1}$ seja escolhido ao final da execução do algoritmo é

$$\frac{1}{n-1} \cdot \frac{n-1}{n} = \frac{1}{n}$$

Por outro lado, a probabilidade do elemento a_n ser o escolhido só depende da última iteração, na qual a_n é escolhido com probabilidade $1/n$, completando a prova. ■

O Exercício 1 pede a generalização deste algoritmo para o caso em que o conjunto de elementos a ser escolhido possuir cardinalidade maior do que um.

5.4 Número de Elementos Satisfazendo uma Propriedade

Dado um fluxo $S = a_1, a_2, \ldots, a_n$, considere o problema de contar o número de elementos de S que satisfazem certa propriedade Π. Por exemplo, pode-se desejar a determinação do número k de elementos de S que são pares, ou maior que certa constante, ou que são raízes de certa equação, ou qualquer propriedade arbitrária de interesse. Naturalmente, uma solução é manter uma variável contadora que é incrementada após a leitura de cada a_i se tal elemento satisfizer a propriedade sendo considerada. Veja o Algoritmo 5.5. O espaço utilizado pelo algoritmo consiste no armazenamento da variável k, sob a hipótese de que a verificação se um elemento satisfaz a propriedade Π é feita em espaço constante. Como $k \leq n$, o espaço é portanto $O(\ell + \log n)$.

Tal complexidade é aceitável para dados massivos. No entanto, considere o problema em que qualquer uso de espaço $\Omega(\log n)$ é inaceitável, ou mais especificamente, procura-se um algoritmo de espaço $O(\ell + \log\log n)$. Tal requerimento pode ser de interesse para aplicações na Internet que serve bilhões de vídeos sob demanda. É de interesse de tais aplicações manter o número de visualizações para cada vídeo. Uma economia em espaço em cada contador pode representar uma grande economia no total.

Algoritmo 5.5 Contagem de elementos satisfazendo Π (espaço $O(\log n)$)

```
função Conta-Elemento(S):
    k ← 0; a ← próximo(S)
    enquanto a ≠ ∅ :
        se a satisfaz Π então
            k ← k + 1
        a ← próximo(S)
    retornar k
```

Primeiramente, note que o valor de k pode ser qualquer um no intervalo $0 \leq k \leq n$. Sendo assim, para manter o algoritmo exato, não é possível usar espaço menor que $\log n$ ou, caso contrário, nem todo valor possível de k poderia ser representado. Assim, para atender ao requerimento de usar espaço $O(\ell + \log\log n)$, devemos aceitar alguma perda. O Algoritmo 5.6 é uma proposta de solução.

De maneira similar ao Algoritmo 5.5, este algoritmo mantém uma variável contadora z, inicialmente nula. Porém, a cada $a_i \in S$ que satisfaz a propriedade de interesse, z não é incrementada sempre. Mais precisamente, o incremento deve ocorrer com probabilidade $1/2^z$. Assim, ao final do algoritmo, $z \leq k$ mas a igualdade ocorre com baixa probabilidade, pois a probabilidade de incremento $1/2^z$ fica exponencialmente menor a medida que z é incrementada, de modo que o valor de z certamente não constitui uma boa estimativa para k. O algoritmo, por outro lado, retorna como estimativa de k o valor $2^z - 1$, o que parece a princípio igualmente não razoável. Basta notar que, ao final do algoritmo, qualquer valor de z no intervalo $0 \leq z \leq k$ é possível e, em particular quando $z = k$, a estimativa $2^k - 1$ é em geral muito maior que k. Embora possível, relembramos que $z = k$ ocorre com baixa probabilidade e, portanto, devemos nos preocupar com os valores mais prováveis da estimativa.

Antes de abordarmos a qualidade desta estimativa formalmente, discutiremos de maneira intuitiva primeiro. Note que, ao final do algoritmo, o retorno $r = 2^z - 1$ é a única saída do algoritmo e toda a informação do fluxo de entrada é perdida. Assim, seja K a variável aleatória representando k, isto é, o número (real) de elementos de S que satisfazem Π. Note que K é de fato uma variável aleatória pois, ao final do algoritmo, é impossível saber com exatidão qual é o valor de k associado a S, uma vez que o fluxo está perdido. Deste modo, muitos valores para K são possíveis, embora nem todos com a mesma probabilidade. Por exemplo, suponha que o retorno do algoritmo tenha sido $r = 1023 = 2^{10} - 1$. Deste modo, sabemos que houve 10 incrementos de z e, por consequência, $k \geq 10$. Por isso, $P(k \leq 9 \mid K = 2^{10} - 1) = 0$.

Já $K = 2^{10} - 1$ é um evento possível, em que

$$P(k = 10 \mid K = 2^{10} - 1) = \frac{1}{2^0} \cdot \frac{1}{2^1} \cdot \cdots \cdot \frac{1}{2^9} = \frac{1}{2^{0+1+\cdots+9}} = \frac{1}{2^{45}}$$

(fazer exercício, pedindo para calcular a probabilidade P(K=1) e P(K=2) de modo que fica claro agora o quão baixa é a probabilidade que $z = k$, mesmo para valores pequenos de z. Por outro lado, $P(k = 10^9 \mid K = 2^{10} - 1)$ também parece ter baixa probabilidade pois, para um número de tal ordem de grandeza, esperaria-se que houvesse muito mais incrementos em z. Assim, procuramos por um valor de K intermediário entre tais extremos, mais provável para o valor real k. A ideia é assumirmos como estimativa para k a esperança de K, isto é, assumir que $k = E[K]$.

A análise formal está estabelecida no Teorema 5.2.

Algoritmo 5.6 Contagem de elementos satisfazendo Π (espaço estimado $O(\log \log n)$)

```
função CONTA-ELEMENTO(S):
    z ← 0; a ← próximo(S)
    enquanto a ≠ ∅ :
        se a satisfaz Π então
            escolher aleatoriamente real p no intervalo 0 ≤ p ≤ 1
            se p ≤ 1/2^z então
                z ← z + 1
        a ← próximo(S)
    retornar 2^z − 1
```

Teorema 5.2. *Sejam $S = a_1, a_2, \ldots, a_n$ o fluxo de entrada dado ao Algoritmo 5.6, $r = 2^z - 1$ o valor de retorno do algoritmo e K a variável aleatória representando o número de elementos de S que satisfazem Π. Assumindo $E[K]$ um estimador de k, o número real de elementos satisfazendo a propriedade em S, então temos que $k = E[K] = r$. Além disso, a complexidade média de espaço do algoritmo é $O(\ell + \log \log n)$.*

Demonstração. Seja INCR a variável aleatória de Bernoulli na qual o sucesso corresponde ao incremento de z quando ocorre a leitura de um a_i que satisfaz a propriedade de interesse Π. Portanto,

$$P(\text{INCR} \mid z = j) = 1/2^j$$

Seja X_j a variável aleatória geométrica que corresponde à ocorrência do primeiro sucesso de INCR a partir do momento em que z se torna igual a j. Em outras palavras, X_j representa o número de elementos que satisfazem a propriedade Π lidos

logo após z se tornar igual a j, e até que z se torne igual a $j+1$. Pela expressão do valor esperado de uma variável geométrica,

$$E[X_j] = \frac{1}{P(\textsc{Incr} \mid z = j)} = \frac{1}{1/2^j} = 2^j$$

Como K é o número de elementos de S satisfazendo a propriedade Π, então

$$K = \sum_{j=0}^{z-1} X_j$$

Logo,

$$\begin{aligned} E[K] &= E\left[\sum_{j=0}^{z-1} X_j\right] \\ &= \sum_{j=0}^{z-1} E[X_j] \\ &= \sum_{j=0}^{z-1} 2^j = 2^z - 1 = r \end{aligned}$$

Pela expressão anterior, temos que $z = \Theta(\log E[K]) = \Theta(\log k) = O(\log n)$. Para armazenar z, é necessário $\Theta(\log z)$ bits, ou seja, espaço $O(\log \log n)$, resultando em um espaço médio de $O(\ell + \log \log n)$. ■

(Adicionar exemplo fictício.)
(Adicionar exemplo real experimental.)

5.5 Número de Elementos Distintos

Seja $S = a_1, a_2, \ldots, a_n$ um fluxo e considere o problema de determinar o número D de elementos distintos de S. Formalmente, o problema é determinar $D = |\{a_1, \ldots, a_n\}|$. Este problema é de interesse em muitas aplicações. Como exemplo, se o fluxo consiste do endereço IP de visitantes a determinado *site*, o número de elementos distintos do fluxo corresponde ao número de visitantes únicos ao *site*, uma importante métrica de popularidade do serviço.

O algoritmo exato, que consiste em armazenar um novo elemento do fluxo apenas se ele é distinto dos demais, seria prático se o número de elementos distintos fosse reduzido. No caso particular em que todos os elementos são distintos, por

exemplo, tal abordagem armazenaria todo o fluxo em memória, o que foge do paradigma que estamos considerando para dados massivos. Os algoritmos apresentados a seguir mantém o espaço sob controle, porém ao custo de produzir uma estimativa aproximada de tal número.

O algoritmo da Contagem Linear

Como vimos, o algoritmo exato de armazenar todos os elementos distintos do fluxo lidos até certo momento não conduz a um algoritmo de dados massivos. Mas prossigamos nesta estratégia desconsiderando o problema do espaço por um momento. A próxima preocupação seria a complexidade de tempo. Quando os elementos distintos são numerosos, verificar se um novo elemento pertence ao conjunto de elementos distintos já lidos, pode levar tempo quadrático no tamanho do fluxo, caso a estrutura de dados não seja bem escolhida. Uma alternativa seria usar uma tabela de dispersão, da seguinte maneira.

Para algum natural L, defina um vetor de ponteiros para listas encadeadas $V[1..L]$, inicialmente nulo, e escolha uma função de dispersão $h : \{0, \ldots, 2^\ell - 1\} \to \{1, \ldots, L\}$. Para cada a_i lido, insira a_i na lista encadeada $V[h(a_i)]$ caso tal elemento não seja encontrado nesta lista. Ao fim da leitura de S, some os tamanhos $|V[i]|$ de todas as listas encadeadas, para $1 \leq i \leq L$. O Algoritmo 5.7 descreve formalmente tal estratégia.

Algoritmo 5.7 Contagem de Elementos Distintos - Paradigma Clássico

função CONTA-ELEMENTOS-DISTINTOS(S):
    defina tabela de dispersão $V[1..L]$ com função de dispersão h
    $a \leftarrow$ *próximo*(S)
    enquanto $a \neq \varnothing$ **:**
        se $V[h(a)]$ não contém a **então**
            insira a em $V[h(a)]$
        $a \leftarrow$ *próximo*(S)
    retornar $\sum_{i=1}^{L} |V[i]|$

Como é conhecido da análise de tabelas de dispersão, a complexidade de tempo de caso médio de cada inserção é de $\Theta(n/L)$, o que se torna tempo constante se $L = \Theta(n)$.

O algoritmo da contagem linear utiliza método análogo, porém sem armazenar os elementos distintos. Ao invés de listas encadeadas, V é um vetor de bits e a inserção do elemento a_i apenas marca a posição $V[h(a_i)]$ com um bit 1. Como elementos repetidos marcam sempre a mesma posição, o número de elementos distintos

é estimado a partir do número de posições não-nulas. Detalharemos esta estratégia em seguida.

Para algum natural L, defina um vetor de bits $V[1..L]$, inicialmente nulo, e escolha uma função de dispersão $h : \{0, \ldots, 2^\ell - 1\} \to \{1, \ldots, L\}$. Para cada a_i lido, defina $V[h(a_i)]$ igual a 1. Ao fim da leitura de S, sejam P o número de posições não-nulas de V e D o número de elementos distintos. Se a função h fosse injetiva (isto é, para todo x, y, $h(x) \neq h(y)$ se $x \neq y$), então P seria igual a D. Mas como este em geral não é o caso, a ideia é obter um estimativa de D em função de P e L. O Algoritmo 5.8 descreve tal ideia, usando os valores de L e $\rho = (L - P)/L$ para a estimativa de D. Note que ρ representa a fração de $V[1..L]$ que possui valores nulos.

Algoritmo 5.8 Contagem de Elementos Distintos - Contagem Linear

função CONTA-ELEMENTOS-DISTINTOS(S):
    defina vetor de bits $V[1..L]$ associado à função de dispersão h
    $V[1..L] \leftarrow 0$; $P \leftarrow 0$
    $a \leftarrow$ *próximo*(S)
    enquanto $a \neq \varnothing$ **:**
        se $V[h(a)] \neq 1$ **então**
            $V[h(a)] \leftarrow 1$; $P \leftarrow P + 1$
        $a \leftarrow$ *próximo*(S)
    $\rho \leftarrow (L - P)/L$
    retornar $-L \ln \rho$

Teorema 5.3. *Seja X_P a variável aleatória que representa o número de posições não-nulas em V ao fim da leitura do fluxo $S = a_1, \ldots, a_n$. Usando como estimativa de P o valor de $E[X_P]$, temos que $D \approx -L \ln \rho$.*

Demonstração. Seja X_i a variável de Bernoulli tal que $X_i = 1$ se $V[i] = 1$ ao fim da leitura de S. Como assumimos que cada uma das D chaves distintas pode ser sorteada uniformemente a cada uma das L posições, temos que cada chave pode ir para a posição i com probabilidade $1/L$, e portanto de não ir com probabilidade $(L - 1)/L$. Logo,

$$P(X_i = 0) = \left(\frac{L-1}{L}\right)^D$$

Usando o fato que $e^r = \lim_{n\to\infty}(1 + r/n)^n$, temos que

$$P(X_i = 1) = 1 - P(X_i = 0) = 1 - \left(\frac{L-1}{L}\right)^D = 1 - \left(\left(1 + \frac{-1}{L}\right)^L\right)^{D/L} \approx 1 - e^{-D/L}$$

Como,

$$X_P = \sum_{i=1}^{L} X_i$$

temos que

$$E[X_P] = \sum_{i=1}^{L} E[X_i] = \sum_{i=1}^{L} P(X_i = 1) \approx L(1 - e^{-D/L})$$

Estimando P por $E[X_P]$, resulta em

$$P \approx L(1 - e^{-D/L})$$

Isolando D, obtemos que

$$D \approx L\ln\left(\frac{L}{L-P}\right) = L\ln\rho^{-1} = -L\ln\rho$$

■

Claramente, a complexidade de espaço do algoritmo é de $\Theta(L)$ e L deve guardar uma relação com n. Com efeito, note que a aproximação acima é sem utilidade quando $L = P$. Naturalmente, a probabilidade de que L se aproxime de P aumenta conforme n aumenta. Assim, é interessante saber para qual n, $E[X_P]$ será igual a L. Note que este é exatamente o problema do colecionador de figurinhas (Seção 4.3.2). Conforme foi visto, se $n \approx L\ln L$, temos que cada posição de V será não-nula. Portanto, o algoritmo produzirá seu efeito se a escolha de L for apropriada, o que ocorre com $n < L\ln L$.

(Colocar um exemplo numérico.)

O algoritmo da Dispersão Mínima

Seja $S = a_1, \ldots, a_n$ uma fonte para a qual se deseja computar o número D de elementos distintos. Seja $h : \{0, \ldots, 2^\ell - 1\} \to [0, 1]$ uma função de dispersão (h é uma função que mapeia cada $a \in \{0, \ldots, 2^\ell - 1\}$ em um real $0 \leq h(a) \leq 1$). A imagem de h é um intervalo contínuo nesta seção por conveniência dos cálculos adiante.

O algoritmo da *dispersão mínima* se baseia na seguinte ideia. Em primeiro lugar, ele computa o valor $h_{\min}$ definido como o mínimo valor da função h sobre todos os elementos de S, isto é,

$$h_{\min} = \min_{1 \leq i \leq n} \{h(a_i)\}$$

Em seguida, observa-se que o valor de $h_{\min}$ é influenciado apenas pelo valor de D e não por n. De fato, como aplicações de h a valores iguais resultam no mesmo

valor, os valores da aplicação de h variam de acordo com o número de elementos distintos de S. Além disso, quanto maior D, maior a chance de que os valores resultantes de h sejam variados. Assim, para que $h_{\min}$ seja um valor extremamente baixo, por exemplo, parece ser razoável esperar que haja um número grande de elementos distintos, já que deve ser pequena a chance de que poucos elementos distintos produzam um valor muito baixo de $h_{\min}$. A partir do valor observado de $h_{\min}$, portanto, o algoritmo visa inferir D. Mais especificamente, tal estimativa é feita assumindo que o valor de $h_{\min}$ é igual ao seu valor esperado. O Algoritmo 5.9 e o Teorema 5.4 fornecem os detalhes desta ideia.

Algoritmo 5.9 Contagem de Elementos Distintos - Dispersão Mínima

função CONTA-ELEMENTOS-DISTINTOS(S):
    $h_{\min} \leftarrow +\infty$
    $a \leftarrow$ *próximo*(S)
    enquanto $a \neq \varnothing$ **:**
        $h_{\min} \leftarrow \min\{h_{\min}, h(a)\}$
        $a \leftarrow$ *próximo*(S)
    retornar $1/h_{\min} - 1$

Teorema 5.4. *Seja S uma dada fonte de dados e h_{min} o valor computado pelo algoritmo associado a S. Seja S' uma fonte arbitrária com o mesmo número D de elementos distintos de S. Seja H_{MIN} a variável aleatória que representa o valor h_{min} computado pelo algoritmo associado a S'. Usando $E[H_{MIN}]$ como estimador de h_{min}, temos que $D = 1/h_{min} - 1$.*

Demonstração. Seja $S_D = d_1, d_2, \ldots, d_D$ uma sequência dos D elementos distintos de S'. Assume-se que a aplicação de h a cada elemento de S_D resulta em um valor real distribuído uniformemente entre 0 e 1. Para $1 \leq i \leq D$, seja X_i a variável que determina o resultado da aplicação em h do i-ésimo elemento de S_D, isto é, $X_i = h(d_i)$. Seja H_{MIN} a variável aleatória que representa o valor $h_{\min}$ computado pelo algoritmo associado a S'. Logo,

$$H_{\text{MIN}} = \min_{1 \leq i \leq D} \{X_i\}$$

Note que $P(X_i \leq k) = k$, para todo $0 \leq k \leq 1$. Como $X_1, X_2, \ldots, X_D$ são indepen-

dentes e igualmente distribuídas, temos que

$$\begin{aligned} P(H_{\text{MIN}} \leq k) &= 1 - P(H_{\text{MIN}} > k) \\ &= 1 - P(X_1, X_2, \ldots, X_D > k) \\ &= 1 - P(X_1 > k)P(X_2 > k) \cdots, P(X_D > k) \\ &= 1 - P(X_1 > k)^D \\ &= 1 - (1 - P(X_1 \leq k))^D \\ &= 1 - (1 - k)^D \end{aligned}$$

Como H_{MIN} é uma variável aleatória contínua, temos que

$$E[H_{\text{MIN}}] = \int_0^1 kf(k)dk$$

onde

$$\begin{aligned} f(k) &= \frac{d}{dk}P(H_{\text{MIN}} \leq k) \\ &= \frac{d}{dk}(1 - (1 - k)^D) = D(1 - k)^{D-1} \end{aligned}$$

Assim, temos que

$$\begin{aligned} E[H_{\text{MIN}}] &= \int_0^1 kf(k)dk \\ &= \int_0^1 kD(1 - k)^{D-1}dk \end{aligned}$$

Fazendo-se a integração por partes, isto é, usando o fato que

$$\int_a^b u(k)\left(\frac{d}{dk}v(k)\right)dk = [u(k)v(k)]_a^b - \int_a^b v(k)\left(\frac{d}{dk}u(k)\right)dk$$

e definindo $u(k) = k$, $\frac{d}{dk}v(k) = D(1 - k)^{D-1}$, $a = 0$ e $b = 1$ (e, por consequência, $v(k) = -(1 - k)^D$ e $\frac{d}{dk}u(k) = 1$), resulta que

$$\begin{aligned} E[H_{\text{MIN}}] &= \int_0^1 kD(1 - k)^{D-1}dk \\ &= \left[-k(1 - k)^D\right]_0^1 - \int_0^1 -(1 - k)^D dk \\ &= -\left[\frac{(1 - k)^{D+1}}{D + 1}\right]_0^1 = \frac{1}{D + 1} \end{aligned}$$

Como $h_{\min}$ é uma estimativa de $E[H_{\text{MIN}}]$, ou seja

$$h_{\min} = E[H_{\text{MIN}}] = \frac{1}{D+1}$$

obtermos, ao isolar D, a expressão

$$D = \frac{1}{h_{\min}} - 1$$

■

Relembre que é assumido que a imagem de h é o intervalo $[0, 1]$. Caso seja preferido utilizar uma função de dispersão $h : \{0, \ldots, 2^\ell - 1\} \to [0, L]$, para alguma constante $L = O(n)$, é possível adaptar diretamente a prova para concluir que

$$D = \frac{L}{h_{\min}} - 1$$

Além disso, é possível mostrar que tal estimativa é uma aproximação do caso discreto, isto é, se h é do tipo $h : \{0, \ldots, 2^\ell - 1\} \to \{0, 1, \ldots, L\}$. Neste caso, como o algoritmo precisa manter apenas o valor $h_{\min}$ sendo computado, usa espaço $\Theta(\log L)$ bits, ou seja, possui complexidade de espaço $O(\log n)$.

O algoritmo *HyperLogLog*

O algoritmo *HyperLogLog* pode ser definido como extensões a partir do algoritmo inicial da contagem probabilística, conforme se segue.

O algoritmo da Contagem Probabilística

Seja $S = a_1, \ldots, a_n$ uma fonte para a qual se deseja computar o número D de elementos distintos, com $|a_i| \leq \ell$ para todo $i = 1, \ldots, n$. Seja $h : \{0, \ldots, 2^\ell - 1\} \to \{0, 1\}^L$ uma função de dispersão (h é uma função que mapeia cada $a \in \{0, 1, \ldots, 2^\ell - 1\}$ em um cadeia binária em L dígitos).

O algoritmo da *contagem probabilística*, assim como o algoritmo da dispersão mínima, também utilizá o resultado de uma função de dispersão para estimar o número de elementos distintos do fluxo, porém de uma outra maneira. A estratégia da contagem probabilística é a seguinte. Para cada elemento a_i do fluxo, computa-se o valor $h(a_i)$ e verifica-se a quantidade $z(h(a_i))$ de símbolos 0 em $h(a_i)$ até a primeira ocorrência de um símbolo 1, lendo-se $h(a_i)$ da esquerda para a direita. Caso $h(a_i)$ seja formado inteiramente por 0's, então $z(h(a_i)) = L$. Computa-se, então, o valor $z_{\max}$ como sendo o maior valor $z(h(a_i))$ encontrado sobre todos os elementos da fonte, isto é,

$$z_{\max} = \max\{z(h(a_i)) \mid 1 \leq i \leq n\}$$

Note que o valor de $z_{\max}$ é influenciado apenas pelo número D de elementos distintos da fonte, e não por n. Mais especificamente, quanto maior o valor de D, maior a chance de que $z_{\max}$ atinja valores maiores. Em particular, para que $z_{\max}$ seja um valor extremamente alto, por exemplo, parece ser razoável esperar que haja um número grande de elementos distintos, já que deve ser pequena a chance de que poucos elementos distintos produzam um valor muito alto de $z_{\max}$. A partir do valor observado de $z_{\max}$, portanto, o algoritmo visa inferir D. Tal estimativa é feita calculando-se a quantidade de elementos distintos de modo que seja possível verificar uma cadeia binária que comece por $z_{\max}$ 0's. O Algoritmo 5.10 e o Teorema 5.5 fornecem os detalhes desta ideia.

Algoritmo 5.10 Contagem de Elementos Distintos - Contagem Probabilística

função CONTA-ELEMENTOS-DISTINTOS(S):
  $z_{\max} \leftarrow -\infty$
  $a \leftarrow$ *próximo*(S)
  enquanto $a \neq \varnothing$ **:**
    $z_{\max} \leftarrow \max\{z_{\max}, z(h(a))\}$
    $a \leftarrow$ *próximo*(S)
  retornar $2^{z_{\max}}$

Teorema 5.5. *Sejam S uma dada fonte de dados com D elementos distintos e $z_{\max}$ o valor computado pelo algoritmo associado a S. Seja $S' = a_1, a_2, \ldots, a_n$ uma fonte arbitrária tal que o primeiro elemento que comece por $z_{\max}$ dígitos zero é a_n. Seja D' a variável aleatória que representa o número de elementos distintos em S'. Usando $E[D']$ como estimador de D, temos que $D = 2^{z_{\max}}$.*

Demonstração. Seja E o evento de que um elemento a de S' comece por $z_{\max}$ zeros. Como cada dígito de a pode ser 0 ou 1 de forma independente dos demais, temos que

$$P(E) = \left(\frac{1}{2}\right)^{z_{\max}} = \frac{1}{2^{z_{\max}}}$$

Seja $S'_{D'} = d_1, d_2, \ldots, d_{D'}$ uma sequência dos D' elementos distintos de S', na mesma ordem com que aparecem em S'. Como a_n é o primeiro elemento que começa por z_{max} dígitos zero, então ele é distinto de todos os demais e, portanto, $d_{D'} = a_n$. Assim, D' pode ser visto como uma variável aleatória geométrica de parâmetro $1/2^{z_{max}}$, pois D' representa o número de tentativas até que E ocorra. Logo,

$$E[D'] = \frac{1}{P(E)} = 2^{z_{\max}}$$

■

Primeira extensão: o algoritmo *LogLog*

O algoritmo *LogLog* é uma melhoria daquele da Contagem Probabilística, apresentado na subseção anterior. Note que a variância que se pode esperar do estimador do número D de elementos distintos, usado no Teorema 5.5, é enorme. Com efeito, as estimativas que podem ser feitas pelo algoritmo anterior são sempre potências de 2, que se distanciam uma da outra cada vez mais à medida que tais potências se tornam maiores. A motivação do algoritmo *LogLog* é diminuir tal variância.

Para tanto, utiliza-se a seguinte ideia. Suponha que, ao invés de apenas um estimador D para o número de elementos distintos, sejam usados k estimadores $D_1, D_2, \ldots, D_k$. Por exemplo, tais estimadores podem ser obtidos individualmente tal como D, porém cada um utilizando uma função de dispersão distinta. Assim, se passarmos a usar como estimador de D a média aritmética $\overline{D}$ de $D_1, D_2, \ldots, D_k$, a variância já diminui por uma razão de k em relação àquela de D. Com efeito, pela Proposição 2.16, temos que

$$\mathrm{Var}(\overline{D}) = \mathrm{Var}\left(\frac{\sum_{i=1}^{k} D_i}{k}\right) = \mathrm{Var}\left(\sum_{i=1}^{k} \frac{1}{k} D_i\right) = \sum_{i=1}^{k} \frac{1}{k^2}\mathrm{Var}(D_i) = \sum_{i=1}^{k} \frac{1}{k^2}\mathrm{Var}(D) = \frac{\mathrm{Var}(D)}{k}$$

No entanto, a estimativa ainda será melhorada em dois aspectos. O primeiro, diz respeito ao peso de cada estimativa D_i na composição da média $\overline{D}$. O segundo, é sobre a eficiência computacional em se executar mais de uma função de dispersão sobre cada elemento da fonte.

Comecemos pelo primeiro aspecto. Note que cada D_i é obtido a partir da expressão 2^{q_i}, para algum natural q_i. Assim, considere o cenário em que $q_1 = q_2 = \cdots = q_{k-1} = a$, para certa constante a, e $q_k = a + 1$. Todas as estimativas, com exceção de uma, resultam em 2^a. No entanto, a estimativa única de 2^{a+1}, pelo fato de ser igual ao dobro de cada uma das outras, põe um grande peso na média aritmética de $\overline{D}$. Assim, pequenas variações nas estimativas individuais ainda resultam, mesmo com a média aritmética, em estimativas médias que se aproximam com maior peso ao valor das maiores estimativas individuais. Assim, basta que apenas uma estimativa entre $D_1, D_2, \ldots, D_k$ falhar, produzindo uma estimativa muito maior que o valor real, que a média $\overline{D}$ será fortemente afetada.

O algoritmo *LogLog* tenta contornar esta fraqueza utilizando uma média alternativa à aritmética. Em seu lugar, o algoritmo utiliza a média geométrica, dada por

$$\overline{D} = \sqrt[k]{D_1 D_2 \cdots D_k}$$

Compare a diferença no impacto de valores atípicos em uma série nas duas médias. Seguindo o exemplo anterior, se $D_1 = D_2 = \cdots = D_{k-1} = a$ e $D_k = 2a$, temos que

a média aritmética

$$\overline{D} = \frac{\left(\sum_{i=1}^{k-1} a\right) + 2a}{k} = \frac{\sum_{i=1}^{k} a}{k} + \frac{a}{k}$$

e verificamos que a média é maior por um fator aditivo de a/k em relação ao que seria se todas as estimativas fossem iguais a a. Tal acréscimo é significativo se $k << a$, o que normalmente é o caso na aplicação de contagem de elementos distintos. Por outro lado, no caso da média geométrica, temos que

$$\overline{D} = \sqrt[k]{\left(\prod_{i=1}^{k-1} a\right) 2a} = \sqrt[k]{2} \sqrt[k]{\prod_{i=1}^{k} a}$$

e, neste caso, a média é maior por um fator multiplicativo de $\sqrt[k]{2}$ em relação ao que seria se todas as estimativas fossem iguais a a. Note que tal fator multiplicativo independente do valor de a e é muito próximo de 1 para valores modestos de k.

Quanto ao segundo aspecto, sobre a necessidade de computar diversas funções de dispersão para cada elemento da fonte e a consequente preocupação no custo computacional, o algoritmo emprega a seguinte ideia. A cada elemento a_i da fonte, é aplicada apenas uma função h de dispersão. Porém, os elementos são particionados em k partes e cada parte produzirá sua própria estimativa, como se toda a entrada fosse reduzida somente àqueles elementos da parte sendo considerada. A partição e a computação de cada estimativa $D_1, D_2, \ldots, D_k$ são conduzidas da seguinte maneira. Escolhe-se $k = 2^b$, para certo natural $b < L$, e considera-se o valor $h(a_i)$ no cálculo apenas da estimativa D_{i+1}, onde i é o natural associado aos b últimos dígitos binários de $h(a_i)$. A Figura 5.2 exemplifica a partição da entrada para a produção de $k = 4$ estimativas distintas para uma fonte com 10 elementos. Para cada elemento a_i da fonte, $h(a_i)$ é uma cadeia com $L = 8$ dígitos binários, dos quais os $b = 2$ últimos são usados para a partição da entrada. Para o exemplo, as estimativas D_1, D_2, D_3, D_4 são calculadas como se mostra na figura.

Esta discussão resulta no Algoritmo 5.11. No algoritmo, se v é uma representação binária, denota-se por v_i o i-ésimo dígito de v lido da esquerda para direita e por $(v)_{10}$ sua representação decimal.

A expressão final que o algoritmo retorna se justifica da seguinte maneira. Em primeiro lugar, note que quando se particiona a entrada em k partes, é esperado que cerca de D/k dos D elementos distintos estejam presentes em cada parte. Portanto, cada estimador D_i, dado pelo valor de $2^{D[i]}$, está estimando a quantidade D/k. Utilizando a média geométrica das estimativas, temos que

$$\frac{D}{k} = \sqrt[k]{\prod_{i=1}^{k} D_i} = \sqrt[k]{\prod_{i=1}^{k} 2^{D[i]}} = 2^{\frac{\sum_{i=1}^{k} D[i]}{k}}$$

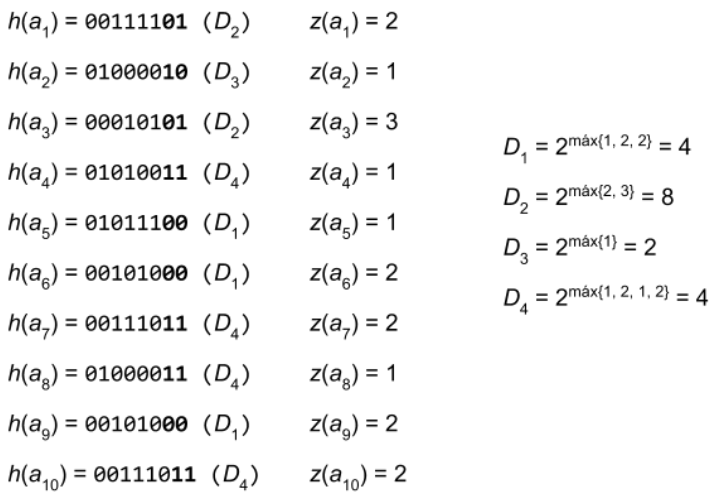

Figura 5.2: Partição da entrada para a produção de $k = 4$ estimativas.

e, portanto,

$$D = k2^{\frac{\sum_{i=1}^{k} D[i]}{k}}$$

Note que a divisão dos D elementos distintos entre os k estimadores não é perfeita. Em verdade, o desequilíbrio causa que certas estimativas serão maiores e outras menores que a real. Em verdade, é possível calcular analiticamente o viés que a análise anterior possui e corrigi-lo através de uma multiplicação de uma constante, que depende exclusivamente do número de partes k. A derivação desta análise está fora do propósito deste texto por ser demasiadamente técnica. A constante α

Algoritmo 5.11 Algoritmo *LogLog*

função LogLog(S):
  definir vetor $D[1..2^b] \leftarrow 0$
  $a \leftarrow$ *próximo*(S)
  enquanto $a \neq \varnothing$ **:**
    $v \leftarrow h(a)$
    $j, a' \leftarrow (v_{L-b+1}v_{L-b+2} \cdots v_L)_{10} + 1, v_1v_2 \cdots v_{L-b}$
    $D[j] \leftarrow \max\{D[j], z(a')\}$
    $a \leftarrow$ *próximo*(S)
  retornar $\alpha k2^{\frac{\sum_{i=1}^{k} D[i]}{k}}$, onde $\alpha = 0{,}79402$

utilizada neste texto pode ser empregada para todo $k \geq 64$ sem notáveis diferenças do valor real.

Segunda extensão: o algoritmo *HyperLogLog*

O algoritmo *HyperLogLog* é uma melhoria do algoritmo *LogLog* apresentado na subseção anterior basicamente por um único aspecto. Análises subsequentes demostraram que a média harmônica possui resultados melhores do que a geométrica.

A média harmônica $\overline{D}$ de $D_1, D_2, \ldots, D_k$ é dada por

$$\overline{D} = \frac{k}{\frac{1}{D_1} + \frac{1}{D_2} + \cdots + \frac{1}{D_k}}$$

Fazendo-se análise análoga à seção anterior, temos que cada estimativa D_i é referente a cerca de D/k elementos, o que resulta no caso da média harmônica que

$$\frac{D}{k} = \frac{k}{\frac{1}{D_1} + \frac{1}{D_2} + \cdots + \frac{1}{D_k}} = k \left(\sum_{i=1}^{k} 2^{-D[i]} \right)^{-1}$$

resultando que

$$D = k^2 \left(\sum_{i=1}^{k} 2^{-D[i]} \right)^{-1}$$

A expressão anterior possui também viés que pode ser calculado, assim como no caso do algoritmo *LogLog*, com dificuldade análoga e também fora dos propósitos do texto. A expressão apresentada no Algoritmo 5.12 serve contanto que $k \geq 128$.

Algoritmo 5.12 Algoritmo *HyperLogLog*

função HyperLogLog(S):
  defina vetor $D[1..2^b] \leftarrow 0$
  $a \leftarrow$ *próximo*(S)
  enquanto $a \neq \varnothing$ **:**
    $v \leftarrow h(a)$
    $j, a' \leftarrow (v_{L-b+1} v_{L-b+2} \cdots v_L)_{10} + 1, v_1 v_2 \cdots v_{L-b}$
    $D[j] \leftarrow \max\{D[j], z(a')\}$
    $a \leftarrow$ *próximo*(S)
  retornar $\alpha_k k^2 \left(\sum_{i=1}^{k} 2^{-D[i]} \right)^{-1}$, onde $\alpha_k = 0{,}7213/(1 + 1{,}079/k)$

Por fim, é necessário observar que as implementações de *HyperLogLog* também trocam de algoritmos de estimação quando as estimativas se aproximam para valores

fora de certo intervalo previsível (isto é, que são muito pequenos ou muito grandes). Tais limites são apontados nos trabalhos originais e estão omitimos neste texto.

5.6 Pertinência ao Fluxo

Seja $S = a_1, a_2, \ldots, a_n$ um fluxo, com $|a_i| \leq \ell$ para todo $1 \leq i \leq n$. O problema de *pertinência ao fluxo* S é aquele de determinar se uma dada cadeia de bits s é um elemento pertencente ao conjunto dos elementos de S, isto é, se $s = a_i$ para algum $1 \leq i \leq n$. Existem inúmeros exemplos nos quais uma solução eficiente para tal problema é de interesse. Note que este problema consiste em uma das operações fundamentais de estrutura de dados, que é aquela de buscar se um dado elemento foi previamente inserido na estrutura de dados.

Como um exemplo moderno de aplicação, podemos citar o problema de executar um *web crawler*. O *crawler* consiste de uma aplicação que, a partir de uma página *web*, acessa outras páginas *web* através de ponteiros (*links*) contidos nesta página. Para cada página encontrada desta forma, o processo se repete de modo recursivo. O objetivo do *crawler* é visitar todas as páginas (ou a maior quantidade possível delas) que podem ser encontradas a partir da página inicial, por tal navegação recursiva. Naturalmente, deseja-se que tal busca evite reprocessar páginas anteriormente já visitadas. Assim, para cada página carregada, é necessário descobrir se ela já foi encontrada anteriormente, através de uma outra sequência de *links*. Uma navegação deste tipo é conduzida, por exemplo, pelos *sites* de busca, que precisam periodicamente varrer a Internet à procura de novas páginas.

Em termos do problema de pertinência a fluxos, podemos modelar a aplicação da seguinte maneira. A página inicial, cujo endereço é inicialmente dado, é carregada e considerada o elemento a_1 do fluxo S. Note que a_1 representa os bits da página propriamente dita, e não apenas a descrição da URL desta página. Isto é útil pois é comum que URLs diferentes carreguem efetivamente o mesmo conteúdo. Para $i \geq 1$, cada nova página a_i carregada produz novos elementos ao fluxo. Cada um deles consiste de uma página que é carregada através dos *links* de a_i, lidos em alguma ordem arbitrária. Naturalmente, tais novos elementos não são necessariamente distintos dos anteriores, já que uma página pode ser descoberta por diferentes sequências de navegações a partir da página inicial. Para evitar inspecionar uma página mais de uma vez, o *crawler* precisa decidir, ao carregar uma nova página $s = a_{i+1}$ através de um *link* de a_i, se $s \in \{a_1, a_2, \ldots, a_i\}$. Se a resposta for afirmativa, os *links* de s são ignorados, uma vez que eles já foram anteriormente considerados. Note que S pode possuir bilhões ou trilhões de elementos, cuja manutenção em memória seria de altíssimo custo.

O filtro de Bloom

Sejam $S = a_1, a_2, \ldots, a_n$ um fluxo, com $|a_i| \leq \ell$ para todo $1 \leq i \leq n$ e s uma dada cadeia de bits. Diremos que $s \in S$ se $s \in \{a_1, a_2, \ldots, a_n\}$. Portanto, o problema da pertinência ao fluxo S é determinar se $s \in S$ ou $s \notin S$.

Uma solução eficiente em tempo é a utilização de uma tabela de acesso direto. Nesta solução, definimos um vetor $V[0..2^\ell - 1]$ de bits, inicialmente nulo. Para cada $i = 1, 2, \ldots, n$, após a leitura de $a_i \in S$, inserimos a_i na tabela atribuindo $V[a_i]$ ao valor 1. Desta forma, para testar se s pertence a S, basta verificar se $V[s] = 1$. Esta ideia é elaborada no Algoritmo 5.13. No algoritmo, s é assumido vir como último elemento de S. Assim, $S = a_1, a_2, \ldots, a_n, s$. Naturalmente, a viabilidade desta solução fica condicionada à disponibilidade de um espaço de 2^ℓ bits.

Algoritmo 5.13 Pertinência a Conjuntos - Paradigma Clássico

```
função PERTINÊNCIA(S):
    defina vetor V[0..2^ℓ − 1] ← 0
    s ← próximo(S); p ← próximo(S)
    enquanto p ≠ ∅ :
        V[s] ← 1
        s ← p; p ← próximo(S)
    retornar V[s] = 1
```

O filtro de Bloom é uma estrutura que resolve o problema de pertinência quando não se dispõe de 2^ℓ bits de espaço, mas $n << 2^\ell$ e é possível dispor de αn bits, para alguma constante α não muito alta. Como exemplo, considere a aplicação de *web crawler* descrita na introdução. Naquela aplicação, o valor de n pode ser considerado da ordem dos bilhões. Para um conjunto de servidores, não é difícil armazenar algumas centenas de bilhões de bits, o que seria espaço da ordem de $100n$. No entanto, como cada a_i consiste de uma página *web*, o valor de ℓ seria o tamanho máximo que uma página pode ter em bits. Tal tamanho já é, como problema inicial, impossível de determinar. Mesmo aplicando um limite máximo de 13 KB $\approx$ 100.000 bits para o tamanho das páginas, o que provavelmente estaria abaixo do tamanho real de diversas páginas, a solução por tabela de acesso direto requer espaço de 2^{100000} bits, o que é impossível.

A ideia do filtro de Bloom é a vetor $V[1..L]$ de bits (chamado de *filtro*), para certa constante $L = O(n)$. Associadas ao filtro, estão k funções de dispersão independentes duas-a-duas, $h_1, \ldots, h_k : \{0, \ldots, 2^\ell - 1\} \to \{1, \ldots, L\}$. O filtro é inicialmente nulo. Para cada $a_i \in S$, inserimos a_i ao filtro, o que é feito através da atribuição de $V[h_j(a_i)]$ ao valor 1 para todo $1 \leq j \leq k$. Para verificar a pertinência de certo dado s a S, o algoritmo responde afirmativamente se $V[h_j(s)] = 1$ para todo $1 \leq$

$j \leq k$ ou, caso contrário, responde NÃO. O Algoritmo 5.14 corresponde a esta ideia geral. A exemplo do Algoritmo 5.13, s é assumido vir como último elemento de $S = a_1, a_2, \ldots, a_n, s$.

Algoritmo 5.14 Pertinência a Conjuntos - Filtro de Bloom

```
função PERTINÊNCIA(S):
    defina vetor V[1..L] ← 0
    s ← próximo(S); p ← próximo(S)
    enquanto p ≠ ∅ :
        para j ← 1 até k :
            V[h_j(s)] ← 1
        s ← p; p ← próximo(S)
    para j ← 1 até k :
        se V[h_j(s)] = 0 então
            retornar F
    retornar V
```

Por construção, se $V[h_j(s)] = 0$ para algum $1 \leq j \leq k$, então claramente $s \notin S$ e a resposta NÃO é correta. No entanto, se $V[h_j(s)] = 1$ para todo $1 \leq j \leq k$, a resposta SIM pode ser equivocada, pois pode ser que parte dos bits 1 encontrados podem ter sido devido a presença de certo $a_p \in S$, enquanto a outra parte devido à presença de outro $a_q \in S$, de modo que $V[h_j(s)] = 1$ para todo $1 \leq j \leq k$ mesmo que $s \notin S$. Portanto, o filtro de Bloom tem como característica a possibilidade de produzir falsos positivos a consultas de pertinência, mas nunca falsos negativos. O Teorema 5.6 determina a probabilidade da ocorrência de um falso positivo.

Teorema 5.6. *Seja $S = a_1, a_2, \ldots, a_n$ uma fonte de dados. Considere o filtro do Bloom associado a funções de dispersão $h_1, h_2, \ldots, h_k$ independentes duas-a-duas com imagem no conjunto $\{1, 2, \ldots, L\}$. Seja* FALSOPOSITIVO *o evento em que, dado uma cadeia $s \notin S$, o algoritmo responde* SIM*, isto é, $V[h_j(s)] = 1$ para todo $1 \leq j \leq k$. Então,*

$$P(\text{FALSOPOSITIVO}) \approx (1 - \mathrm{e}^{-nk/L})^k$$

Além disso, mantendo-se fixo a razão n/L, a probabilidade acima é minimizada para

$$k = \frac{L\ln(2)}{n}$$

correspondendo a

$$P(\text{FALSOPOSITIVO}) \approx (0.6185)^{\frac{L}{n}}$$

Demonstração. Sob a hipótese de que as funções de dispersão são independentes,

$$\begin{aligned} P(\textsc{FalsoPositivo}) &= P(V[h_j(s)] = 1 \text{ para todo } 1 \leq j \leq k) \\ &= P(V[h_1(s)] = 1)P(V[h_2(s)] = 1) \cdots P(V[h_k(s)] = 1) \\ &= P(V[p] = 1)^k, \text{ para algum } 1 \leq p \leq L \end{aligned}$$

Note que para certo $1 \leq j \leq k$ e $1 \leq i \leq n$, temos que

$$P(h_j(a_i) \neq p) = \frac{L-1}{L} = 1 - \frac{1}{L}$$

Como para que certa posição p seja tal que $V[p] = 0$ é necessário que $h_j(a_i) \neq p$ para todo $1 \leq j \leq k$, $1 \leq i \leq n$, temos que

$$\begin{aligned} P(V[p] = 1) &= 1 - P(V[p] = 0) \\ &= 1 - P(h_j(a_i) \neq p \text{ para todo } 1 \leq j \leq k,\, 1 \leq i \leq n) \\ &= 1 - \left(1 - \frac{1}{L}\right)^{nk} \end{aligned}$$

Usando o fato que $e^r = \lim_{n\to\infty}(1 + r/n)^n$, temos que

$$\begin{aligned} P(V[p] = 1) &= 1 - \left(\left(1 + \frac{-1}{L}\right)^L\right)^{nk/L} \\ &\approx 1 - \mathrm{e}^{-nk/L} \end{aligned}$$

Portanto,

$$P(\textsc{FalsoPositivo}) \approx (1 - \mathrm{e}^{-nk/L})^k$$

Passemos agora à questão relativa ao número k de funções de dispersão que minimiza a probabilidade de falso positivo, mantendo-se fixo n/L. Para tanto, derivando a expressão $P(\textsc{FalsoPositivo})$ com respeito a k e igualando-a a zero, temos uma equação em k que conduz ao valor mínimo. Para simplificar um pouco as contas, contudo, utiliza-se do fato de que para funções quaisquer $f(k), g(k)$ tais que

$$f(k) = \mathrm{e}^{g(k)}$$

temos que

$$\frac{\mathrm{d}f(k)}{\mathrm{d}k} = \mathrm{e}^{g(k)}\frac{\mathrm{d}g(k)}{\mathrm{d}k}$$

e, como $e^{g(k)}$ não se anula, a derivada de $f(k)$ se anula exatamente nos mesmos pontos para os quais a derivada de $g(k)$ se anula. Aplicando tal observação, com $f(k) = P(\text{FalsoPositivo})$ e $g(k) = \ln P(\text{FalsoPositivo}) = k\ln(1 - e^{-nk/L})$, queremos determinar as raízes de

$$\frac{dg(k)}{dk} = \ln(1 - e^{-nk/L}) + \frac{nk}{L} \cdot \frac{e^{-nk/L}}{1 - e^{-nk/L}}$$

É fácil verificar que

$$k = \frac{L\ln(2)}{n}$$

é uma raiz da função acima. Pode-se mostrar também (Exercício 2) que este é o ponto de mínimo global. Naturalmente, o número ótimo de funções de dispersão deve ser um valor inteiro próximo daquele de k calculado acima. Portanto, para tal valor de k,

$$\begin{aligned} P(\text{FalsoPositivo}) &\approx \left(\frac{1}{2}\right)^{\frac{L\ln(2)}{n}} \\ &\approx (0.6185)^{\frac{L}{n}} \end{aligned}$$

■

Assim, para $L = 10n$, a probabilidade de se obter falsos positivos é de $0,82\%$.

5.7 Frequência de Elementos

Seja $S = a_1, a_2, \ldots, a_n$ um fluxo, com $|a_i| = \ell$ para todo $1 \leq i \leq n$. Nesta seção, trataremos de um problema mais geral que aquele de pertinência. Estaremos interessados em determinar a frequência de um elemento, ou seja, a quantidade de elementos do fluxo iguais a dado elemento de consulta ou de algum elemento especial do fluxo. Dentre as aplicações deste tipo de problema, podemos citar aquela relacionada a um processo de votação. Em tal processo, a partir de um fluxo com os votos, cada voto consistindo da identificação de um candidato dentre $m = 2^\ell$ possíveis, deseja-se saber a frequência com que os candidatos foram votados. Em particular, pode-se desejar determinar ou o candidato mais votado, ou aqueles que foram votados acima de um limiar de número de votos, ou se há um candidato que tenha recebido mais da metade dos votos, entre outros problemas. A próxima seção é dedicada a este último problema.

O algoritmo do elemento majoritário

O primeiro problema desta categoria, que chamaremos de *problema do elemento majoritário*, será aquele para decidir se algum elemento do fluxo recebeu mais da metade dos votos e, no caso afirmativo, determinar tal elemento. Este problema já foi estudado na Seção 3.3.2, no contexto de algoritmos randomizados. No contexto deste capítulo, os votos são consideramos uma fonte de dados, para a qual é impraticável (ou inconveniente) manter seus elementos todos em memória. Em primeiro lugar, mostramos que qualquer algoritmo determinístico que resolva este problema em uma única leitura do fluxo requer espaço $\Omega(\min(n, m))$, o que torna impraticável a sua aplicação às fontes de dados massivas.

Teorema 5.7. *Seja $S = a_1, a_2, \ldots, a_n$ um fluxo, com $0 \leq a_i < m = 2^\ell$ para todo $1 \leq i \leq n$. Qualquer algoritmo determinístico que resolva o problema do elemento majoritário em uma única leitura do fluxo requer espaço $\Omega(\min(n, m))$.*

Demonstração. Seja $\mathcal{C}_k$ o conjunto de todos os fluxos com k elementos. Seja $f(S')$ a função que mapeia um fluxo $S' \in \mathcal{C}_k$ ao conjunto de seus elementos distintos. Isto é, os elementos do contradomínio da função f são os subconjuntos de $\mathcal{S} = \{0, 1, \ldots, m-1\}$. Se $k \geq m$, qualquer elemento de $\mathcal{S}$ é imagem de algum fluxo de $\mathcal{C}_k$. Logo, o conjunto $\mathrm{Im}(f)$ imagem de f é tal que $|\mathrm{Im}(f)| = 2^m$. Se $k < m$, então nem todo subconjunto de $\mathcal{S}$ pode ser mapeado por algum elemento de $\mathcal{C}_k$. Em especial, por exemplo, o próprio conjunto $\mathcal{S}$ não pode ser mapeado, pois não há número suficiente de elementos na fonte. Neste caso, temos que

$$|\mathrm{Im}(f)| = \sum_{i=1}^{k} \binom{m}{i} \geq \sum_{i=1}^{k} \binom{k}{i} = 2^k - 1 = \Theta(2^k)$$

Portanto, para $k = \lceil n/2 \rceil$, temos que

$$|\mathrm{Im}(f)| = \Omega(\min(2^{n/2}, 2^m))$$

Para se representar de maneira distinta todos os elementos de $\mathrm{Im}(f)$, é necessário uma memória de $\Omega(\min(n, m))$ bits. Logo, para qualquer algoritmo que utilize espaço M de memória com $M = o(\min(n, m))$ bits, existiriam dois fluxos $S'_1, S'_2 \in \mathcal{C}_k$, com $f(S'_1) \neq f(S'_2)$, para os quais o algoritmo representaria indistintamente $f(S'_1)$ e $f(S'_2)$ em M. Como $f(S'_1) \neq f(S'_2)$, sem perda de generalidade, seja c um elemento que pertença a $f(S'_1)$ mas não a $f(S'_2)$. Construa o fluxo S_1 (resp. S_2) com n elementos a partir de S'_1 (resp. S'_2) pelo acréscimo de $a_{k+1} = a_{k+2} = \cdots = a_n = c$. O elemento c seria majoritário em S_1, mas não em S'_2. Como com espaço M o algoritmo não consegue distinguir $f(S'_1)$ de $f(S'_2)$, o algoritmo não poderia diferenciar ambos os possíveis resultados sobre a existência de elemento majoritário em S_1 e S_2 sem voltar a consultar os elementos do fluxo. ■

Como consequência do Teorema 5.7, os algoritmos determinísticos para se resolver o problema do elemento majoritário em única leitura de um fluxo são inviáveis se n, m são ambos muito grandes. No entanto, há um algoritmo determinístico com espaço $O(\log n + \ell)$ se for permitido relaxar o requerimento de que o algoritmo diferencie o caso de haver ou não um elemento majoritário. O único requerimento para o algoritmo é que, caso exista um elemento majoritário, que ele seja determinado corretamente. Caso não exista, o algoritmo poderia reportar qualquer elemento. Em suma, há considerável ganho de espaço se é admissível falsos positivos na solução do problema, mas não falsos negativos.

O algoritmo é como segue. Inicializa-se a variável s com o valor 0 e define-se a variável c como nulo. O papel da variável c é guardar, a todo momento, o elemento que será eleito como majoritário ao fim do algoritmo. O valor de s pode ser entendido, informalmente por ora, como o grau de confiança que o algoritmo tem sobre c ser o elemento correto. O algoritmo poderá mudar o valor de c ao longo de sua execução, mas ao final, sempre reportará um valor c como majoritário. A política da escolha do elemento majoritário é dada a seguir. Para cada elemento a_i da fonte, incrementa-se s se $c = a_i$, isto é, o novo elemento lido é igual ao candidato corrente a majoritário. Caso $c \neq a_i$, então decrementa-se s se $s > 0$. Se s for nulo, o algoritmo "perde a confiança" de que o valor vigente de c é o candidato certo e troca para o novo lido, isto é, faz-se $c = a_i$ e $s = 1$. Ao final, reporta-se c como elemento majoritário. O Algoritmo 5.15 descreve mais precisamente este algoritmo e o Teorema 5.8 se preocupa com sua correção. Como c é representável por ℓ bits e $s = O(n)$, segue que a complexidade de espaço do algoritmo é $O(\log n + \ell)$.

Algoritmo 5.15 Elemento Majoritário

```
função Elemento-Majoritário(S):
    c, s ← Nulo, 0
    a ← próximo(S)
    enquanto a ≠ ∅ :
        se c = a então
            s ← s + 1
        senão
            se s > 0 então
                s ← s − 1
            senão
                c, s ← a, 1
        a ← próximo(S)
    retornar c
```

Teorema 5.8. *Seja $S = a_1, a_2, \ldots, a_n$ um fluxo, com $|a_i| = \ell$ para todo $1 \leq i \leq n$, onde cada a_i é pertence a $\{0, 1, \ldots, m-1\}$, com $m = 2^\ell$. O Algoritmo 5.15 retorna o elemento majoritário caso exista um.*

Demonstração. Para a prova, vejamos cada novo elemento da fonte como um voto e precisamos verificar, ao final, se há um candidato eleito majoritariamente. A correção do algoritmo pode ser verificada da seguinte maneira. Primeiro, afirmamos que a seguinte proposição é válida após a inicialização do algoritmo para $i = 0$ e ao término da i-ésima iteração para todo $1 \leq i \leq n$:

$$\text{nvotos}(p, i) \leq \begin{cases} \frac{i+s}{2} & \text{, se } p = c \\ \frac{i-s}{2} & \text{, se } p \neq c \end{cases}$$

onde $\text{nvotos}(p, i)$ consiste do número de votos que o candidato p recebe nos primeiros i elementos de S. Sob a hipótese da validade desta proposição, temos que, ao final do algoritmo, vale que

$$\text{nvotos}(p, n) \leq (n - s)/2$$

para todo $p \neq c$. Como $s \geq 0$, isto equivale dizer que todo candidato distinto de c não recebeu votos da maioria. Logo, ou há um candidato votado pela maioria e este é precisamente c, ou simplesmente não há, o que comprova a correção. Falta apenas comprovar a validade da proposição.

A prova é feita por indução em i. Para $i = 0$, temos que $\text{nvotos}(p, 0) = (i-s)/2 = (i + s)/2 = 0$ para qualquer p, pois $i = s = 0$ após a inicialização, o que torna a proposição trivialmente válida. Se $i > 0$, suponha que a afirmação seja válida para todo $0 \leq i' < i$. Para o caso geral, note que a proposição faz duas subafirmações, uma para o candidato corrente c e outra para os demais. Além disso, as variáveis c, s são modificadas de três diferentes maneiras, conforme se (i) $c = a_i$, (ii) $c \neq a_i$ e $s > 0$, e (iii) $c \neq a_i$ e $s = 0$. Assim, a prova consiste de mostrar que cada subafirmativa é válida em cada um dos casos, o que totaliza 6 verificações. Serão apresentadas duas delas a seguir.

- para $c = a_i$: neste caso, a variável s é incrementada. Assim, por hipótese de indução, ao final da i'-ésima iteração, com $i' < i$, era válido que

$$\text{nvotos}(p, i') \leq \begin{cases} \frac{i'+s'}{2} & \text{, se } p = c \\ \frac{i'-s'}{2} & \text{, se } p \neq c \end{cases}$$

onde s' representa o valor de s ao fim daquela iteração. Para o caso particular da iteração anterior, temos que

  - se $p = c$:

$$\text{nvotos}(p, i) = \text{nvotos}(p, i-1) + 1 \leq \frac{(i-1) + (s-1)}{2} + 1 = \frac{i+s}{2}$$

– se $p \neq c$:

$$\text{nvotos}(p, i) = \text{nvotos}(p, i-1) \leq \frac{(i-1)-(s-1)}{2} = \frac{i-s}{2}$$

As demais quatro verificações podem ser realizadas de forma análoga (Exercício 3) ■

5.8 Semelhança entre Conjuntos

Nesta seção, discutiremos o problema de determinar, dados os conjuntos A e B, o quão semelhantes eles são entre si. A medida de semelhança utilizada é a proporção de elementos comuns entre A e B. Mais especificamente, a *semelhança* entre A e B, conhecida como *coeficiente de semelhança de Jaccard* e denotada por $J(A, B)$, é definida como

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Pode-se considerar diversas aplicações relacionadas ao problema. Originalmente, a determinação de semelhança foi empregada na detecção de páginas *web* que eram cópias de outras, a menos de pequenas modificações. Para isso, cada página era representada por seu conjunto de palavras e considerava-se que duas páginas eram semelhantes o suficiente para serem consideradas cópias se a semelhança entre os conjuntos de palavras correspondentes ultrapassava certo limiar. Aplicações análogas na detecção de imagens similares são encontradas na literatura. Em aplicações desta natureza, os conjuntos a terem sua semelhança medidas podem ser de cardinalidade considerável. Nesta seção, assume-se que tais cardinalidades são arbitrariamente grandes, de modo que os elementos de cada conjunto são lidos como fonte de dados.

Sejam $S_A = a_1, \ldots, a_r$ e $S_B = b_1, \ldots, b_s$ fontes para as quais se deseja computar $J(A, B)$, onde $A = \{a_1, \ldots, a_r\}$ e $B = \{b_1, \ldots, b_s\}$. Sejam $n = \max\{r, s\}$ e ℓ o maior tamanho de elemento em bits. No paradigma clássico, pode-se computar $J(A, B)$ em tempo $O(n \log n)$ e espaço $O(n\ell)$. Porém, em tais algoritmos, todos os elementos precisam ser guardados ao mesmo tempo em memória. Há algoritmos de tempo $O(n)$ e espaço $O(2^\ell)$ para os quais isto não é necessário, mas o espaço requerido é, em geral, proibitivo (ver Exercício 4). A solução apresentada a seguir produz uma estimativa em complexidade de tempo $O(n)$, espaço constante, e requer apenas uma leitura de cada elemento.

A técnica a ser apresentada se assemelha àquela empregada no algoritmo de dispersão mínima, apresentado na Seção 5.5. Seja h uma função de dispersão injetora (isto é, $h(x) \neq h(y)$ se $x \neq y$). A imagem de h é considerada ser um valor armazenável em um número constante de palavras de máquina (ou seja, de espaço

constante). O algoritmo para semelhança se baseia no valor $h^A_{\min}$ (resp. $h^B_{\min}$) definido como o mínimo valor da função h sobre todos os elementos de S_A (resp. S_B). Isto é,

$$h^A_{\min} = \min_{1\leq i\leq r}\{h(a_i)\} \quad \text{e} \quad h^B_{\min} = \min_{1\leq i\leq s}\{h(b_i)\} \tag{5.1}$$

Em seguida, compara-se o valor de $h^A_{\min}$ e $h^B_{\min}$. Se $A = B$, então certamente $h^A_{\min} = h^B_{\min}$. Caso contrário, ainda pode ocorrer de $h^A_{\min} = h^B_{\min}$ mesmo quando $A \neq B$. E o ponto fundamental da ideia é a determinação da probabilidade com que tal evento ocorre.

Lema 5.9. *Sejam $h^A_{\min}$ e $h^B_{\min}$ determinados como em (5.1). Temos que*

$$P(h^A_{\min} = h^B_{\min}) = J(A, B)$$

Demonstração. Primeiramente, seja $h_{\min}$ o menor valor de dispersão sobre todos os elementos de A e B. Ou seja,

$$h_{\min} = \min_{a\in A\cup B}\{h(a)\}$$

Note que

$$h_{\min} = \min\{h^A_{\min}, h^B_{\min}\}$$

Seja $a_{\min} \in A \cup B$ tal que $h(a_{\min}) = h_{\min}$. Observe que $h^A_{\min} = h^B_{\min}$ se, e somente se,

$$h^A_{\min} = h^B_{\min} = h_{\min}$$

que por sua vez ocorre se, e somente se,

$$a_{\min} \in A \cap B$$

Portanto, a probabilidade de que $h^A_{\min} = h^B_{\min}$ é a mesma daquela que um elemento especial ($a_{\min}$) de $A \cup B$ esteja em ambos A e B. Formalmente,

$$P(h^A_{\min} = h^B_{\min}) = P(a_{\min} \in A \cap B) = \frac{|A \cap B|}{|A \cup B|} = J(A, B)$$

■

Seja X_h a variável aleatória de Bernoulli tal que, dadas fontes A, B, ocorre que $X_h = 1$ se $h^A_{\min} = h^B_{\min}$. Portanto, pelo Lema 5.9, temos que $X_h = 1$ com probabilidade $p = J(A, B)$ e, assim, o valor esperado de $E[X_h]$ é p. Porém, a variância de X_h é muita alta, pois em cada medição, os únicos valores possíveis de X_h são 0 ou 1, e nada entre eles. Assim, o algoritmo que virá a seguir emprega a ideia de utilizar k funções $h_1, h_2, \ldots, h_k$ de dispersão e, para cada uma, medir o valor

da variável aleatória X_{h_i} e usar como estimador $\overline{J}$ de $J(A,B)$ a média aritmética de $X_{h_1}, X_{h_2}, \ldots, X_{h_k}$. Em outras palavras,

$$\overline{J} = \frac{\sum_{i=1}^{k} X_{h_i}}{k}$$

Enquanto $\mathrm{Var}(X_h)$ corresponde à variância no caso de uma única função de dispersão h, para o estimador $\overline{J}$ temos que

$$\mathrm{Var}(\overline{J}) = \mathrm{Var}\left(\frac{\sum_{i=1}^{k} X_{h_i}}{k}\right) = \mathrm{Var}\left(\sum_{i=1}^{k} \frac{1}{k} X_{h_i}\right) = \sum_{i=1}^{k} \frac{1}{k^2}\mathrm{Var}(X_{h_i}) = \sum_{i=1}^{k} \frac{1}{k^2}\mathrm{Var}(X_h) = \frac{\mathrm{Var}(X}{k}$$

O Algoritmo 5.16 formaliza a discussão anterior para a semelhança de conjuntos pelo coeficiente de Jaccard. A tupla $((h_1)^A_{\min}, (h_2)^A_{\min}, \ldots, (h_k)^A_{\min})$ com os valores de $h^A_{\min}$ para cada função de dispersão $h = h_1, h_2, \ldots, h_k$ com respeito ao conjunto A é chamada de *assinatura* de A.

Algoritmo 5.16 Semelhança de Conjuntos – Coeficiente de Jaccard

```
função Obtém-Assinatura(S):
    definir vetor H_min[1..k] ← +∞
    a ← próximo(S)
    enquanto a ≠ ∅ :
        para i ← 1 até k :
            H_min[i] ← min{H_min[i], h_i(a)}
        a ← próximo(S)
    retornar H_min

função Semelhança(S_A, S_B):
    H^A_min ← Obtém-Assinatura(S_A)
    H^B_min ← Obtém-Assinatura(S_B)
    c ← 0
    para i ← 1 até k :
        se H^A_min[i] = H^B_min[i] então
            c ← c + 1
    retornar c/k
```

5.9 Programas em Python

5.10 Exercícios

5.1 Ajuste o Exemplo 2 para que o algoritmo escolha k elementos de S com distribuição uniforme, ao invés de apenas um.

5.2 Prove que, para filtros de Bloom onde se mantém n/L fixo, o valor de $k = \frac{L\ln(2)}{n}$ é um ótimo global para a expressão da probabilidade de falsos positivos.

5.3 Termine a prova de correção do algoritmo do elemento majoritário (Seção 5.7), estendendo a análise do caso geral da indução para as outras 4 verificações que ficaram pendentes, conforme discutido na prova.

5.4 Sejam A e B conjuntos de cardinalidade n com elementos arbitrários. Considere que o tamanho de cada elemento de A ou B possua no máximo ℓ bits. Elabore algoritmos para determinar $J(A, B)$ com as seguintes complexidades:

(i) tempo $O(n \log n)$ e espaço $O(n\ell)$

(ii) tempo $O(n)$ e espaço $O(2^\ell)$