Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Um, dois, três, ... muitos, ...

muitos mesmo

William Hideki Kondo

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisor: José Coelho de Pina

São Paulo
25 de Fevereiro de 2022

Resumo

William Hideki Kondo. **Um, dois, três, ... muitos, ...: *muitos mesmo***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Contagem distinta é o problema de se encontrar o número de elementos distintos em um fluxo de dados com repetição de elementos. A solução trivial, que insere os dados em um tabela, tem um consumo de espaço linear e é inviável para aplicações com alto volume de dados. Algoritmos probabilísticos resolvem esse problema trocando a exatidão da contagem por uma grande redução do consumo de espaço. Então, este texto apresentará soluções probabilísticas para a contagem distinta.

Palavras-chave: Contagem distinta. Aproximação. Fluxo de dados.

Abstract

William Hideki Kondo. **One, two, three, … many, …: *definitely many***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Count-distinct is the problem of finding the number of distinct elements in a data stream with repeated elements. The trivial solution, that inserts the data in a table, has a linear space consumption and it is impracticable for high volume data applications. Probabilistic algorithms solve this problem by exchanging the exact count for a great space consumption reduction. Therefore, this text will introduce probabilistic solutions for count-distinct.

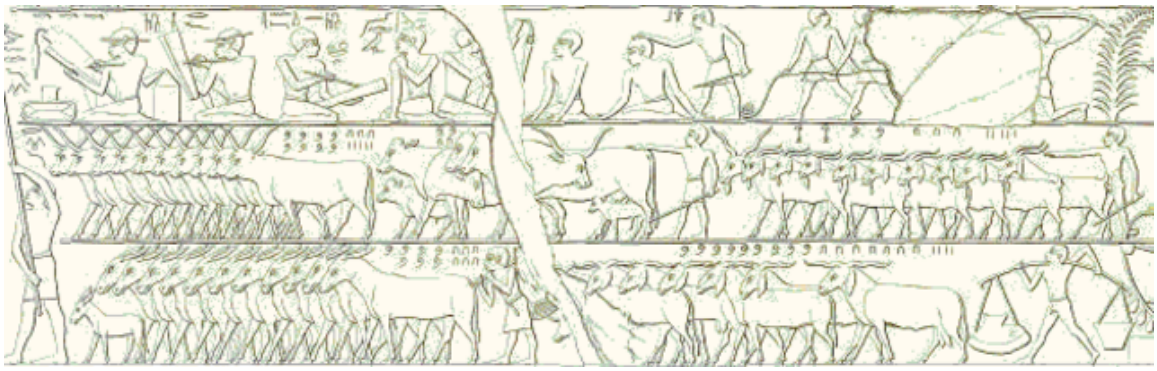
Keywords: Count-distinct. Approximation. Data stream.

Sumário

Capítulo 1

Introdução

Desde a pré-história, o ser humano tem a experiência de contar. Naquela época, o homem contava pequenas quantidades, como quantas pessoas tinham em seu bando ou quanto de alimento ele deveria coletar para sobreviver. Com o passar do tempo, o sistema numérico foi inventado para que pudéssemos trabalhar com valores cada vez maiores.



Quando pensávamos que contar não pudesse ficar mais difícil, os computadores surgem, abrindo novas discussões sobre a contagem. Entre elas, estavam como armazenar números em uma máquina, como fazer contas nesses aparelhos e até quanto poderíamos contar com a ajuda deles. E poucas décadas após essa invenção, a Internet é criada.

Passamos, portanto, a ficar interessados em monitorar essa rede de computadores e para isso, tínhamos que descobrir quantas máquinas diferentes estavam acessando essa rede ou quem eram os aparelhos responsáveis pelo maior fluxo de dados. O principal desafio que surgiu nesse contexto, foi o alto volume de dados que precisavam ser processados em tempo real. E armazenar todos os dados localmente para em seguida, analisá-los deixou de ser viável devido à lentidão desse processo e ao elevado consumo de memória.

Estruturas de dados e algoritmos probabilísticos são uma forma de tentar contornar essa grande quantidade de informação. A ideia central é sacrificar a exatidão da resposta com o objetivo de consumir menos memória e tempo de processamento. Problemas que podem se beneficiar com soluções probabilísticas são, por exemplo, monitorar quantos usuários diferentes acessaram um site em um dado dia, contar quantas palavras distintas

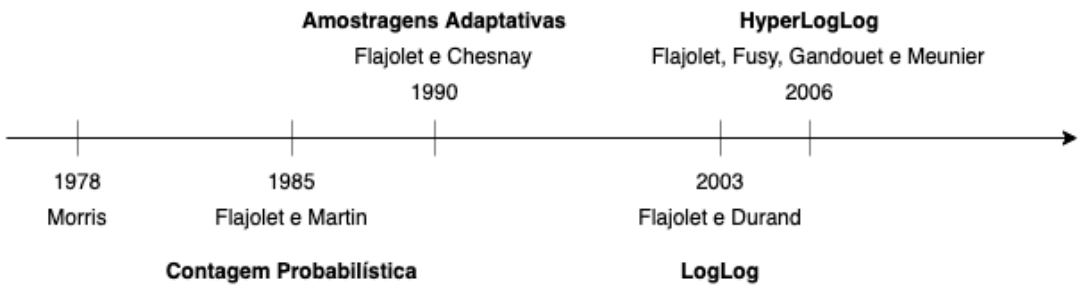
foram pesquisadas na última hora em uma plataforma de varejo ou armazenar quantas visualizações distintas um artigo teve.

Uma característica comum desses problemas é que muitas vezes, suas respostas são métricas a serem utilizadas por outros sistemas com o intuito de se identificar falhas ou pontos de melhoria. No caso do monitoramento de quantas pessoas visitaram uma página web, uma forte queda nas visualizações pode ser um indício que um serviço esteja fora do ar. Nesse sentido, essas métricas não precisam ter necessariamente uma precisão de 100%, podendo apresentar um pequeno erro desde que sejam rapidamente computadas e leves de se armazenarem.

Por volta da década de 1970, Robert Morris tentou contar eventos cujo número de ocorrências não cabia na memória dos computadores da época (Morris, 1978). O algoritmo proposto por ele serviu de inspiração para que outros autores conseguissem resolver problemas mais desafiadores, como a contagem distinta aproximada, cujo objetivo é estimar a quantidade de elementos distintos em um fluxo de dados.

Alguns anos após Morris publicar o trabalho dele, a Contagem Probabilística, o primeiro algoritmo probabilístico que resolvia a contagem distinta aproximada, foi criada (Flajolet e Martin, 1985). E a partir desse algoritmo, as soluções foram passando por melhorias, das quais podemos destacar a redução do consumo de memória, aumento da precisão e até mesmo o desenvolvimento de técnicas de demonstração que simplificavam o entendimento da razão desses algoritmos funcionarem.

Pouco tempo depois, as Amostragens Adaptativas, algoritmo que corrigia limitações da solução anterior, foram desenvolvidas (Flajolet, 1990). Após uma década, a estrutura de dados LogLog é concebida, apresentando uma grande redução no consumo de memória (Durand e Flajolet, 2003). E logo em seguida, o HyperLogLog, versão aperfeiçoada do LogLog, veio a público e se tornou uma das estruturas mais utilizadas para se resolver o problema da contagem distinta aproximada (Flajolet, Fusy *et al.*, 2007). Este texto, portanto, tem o objetivo de passar por essas soluções e apresentar comentários pertinentes de cada uma.



1.1 Probabilidade

Neste texto, serão discutidas soluções probabilísticas de problemas relacionados à contagem. Para entendê-las, alguns conceitos relacionados à Estatística precisam estar claros.

1.2 Variância

Seja X uma variável aleatória. Então,

$$\mathbb{V}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

1.3 Desigualdade de Markov

Sejam X uma variável aleatória e $\alpha > 0$ um número real. Então,

$$\mathbb{P}(X \geq \alpha) \leq \frac{\mathbb{E}[X]}{\alpha^2}$$

1.4 Desigualdade de Chebyshev

Seja X uma variável aleatória com valor esperado μ finito e variância σ^2 finita e diferente de zero. Assim, para todo número real $k > 0$,

$$\mathbb{P}(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

Outro modo de se escrever a desigualdade acima é

$$\mathbb{P}(|X - \mu| \geq k) \leq \frac{\sigma^2}{k^2}$$

1.5 Transformação de Mellin

A transformação de Mellin para uma função *real* $f(x)$ definida para $x \in \mathbb{R}$ e $x \geq 0$ é uma função *complexa* $f^*(x)$ tal que:

$$f^*(s) := M\big[f(x); s\big] = \int_0^\infty f(x)^{s-1} dx \,.$$

Capítulo 2

Contagens aproximadas

Na década de 1970, Robert Morris (Morris, 1978) estudou o problema de contar rapidamente *trigramas* cujas frequências podiam chegar até 130 mil. Esses trigramas eram trios de caracteres que ocorriam em textos.

O objetivo, então, era utilizar as contagens de trigramas na implementação de corretores ortográficos estatísticos (Morris e Cherry, 1975) para editores de textos (McMahon *et al.*, 1978) que acompanhariam as distribuições do sistema operacional Unix dos laboratórios da Bell (Lumbroso, 2018). Contagens similares têm aplicação no projeto de compressores de textos estatísticos (Bell *et al.*, 1990).

Na época, Morris utilizava um computador PDP11 da DEC com memória de 8 bits para realizar as contagens. Com 8 bits, somos capazes de representar ou *contar* de 0 até 255. Em geral, uma memória de n bits pode armazenar números inteiros entre 0 e $2^n - 1$.

Devido à limitação de espaço e ao desejo de eficiência, Morris projetou um algoritmo para realizar estimativas ou *contagens aproximadas*, uma vez que não era possível manter as frequências *exatas* dos trigramas. Com seu algoritmo, Morris trouxe aleatorização e probabilidade para o centro da discussão. E para tornarmos essa discussão mais precisa, consideraremos o problema de determinar um número **aproximado** de elementos de um dado **conjunto** $\mathbb{M}$.

> **Problema** da ContagemAproximada($\mathbb{M}, k$): dado um conjunto $\mathbb{M} = \{e_1, e_2, \ldots, e_n\}$, encontrar estimador $\hat{n}$ para o número n de elementos em $\mathbb{M}$ usando não mais que k bits.

Quando $\mathbb{M}$ é conhecido de antemão, teremos um problema que é dito **estático** ou ***offline***. Já em problemas **dinâmicos** ou ***online***, receberemos uma sequência de operações sobre $\mathbb{M}$ que não são previamente conhecidas e devem ser realizadas uma após a outra. Entre essas operações, existirão as **consultas** (*queries*) que podem ser, por exemplo, encontrar o número de elementos em $\mathbb{M}$. Teremos, também, as **atualizações** (*updates*) que nesse capítulo, serão modificações de $\mathbb{M}$ somente por meio de inserções de elementos.

Nesse sentido, o problema estudado por Morris pode ser considerado **incremental** (não há remoções de elementos), e dessa forma, o conjunto $\mathbb{M}$ pode ser visto como sendo um

fluxo $x_1, x_2, \ldots, x_t, \ldots$ de elementos, e as consultas seriam "Quantos elementos o conjunto $\mathbb{M}$ possui **neste instante**?" ou "Quantos elementos já passaram por este fluxo".

No entanto, estudaremos detalhadamente nesse texto, as versões estáticas dos problemas apresentados, uma vez que, as análises dos algoritmos que resolvem esses problemas se baseiam nessas versões. E veremos que obter a versão *online* a partir da *offline* é uma tarefa simples, e assim, as aplicações e simulações desses algoritmos serão baseadas em versões dinâmicas.

Na próxima seção, veremos com mais detalhes o algoritmo de Morris que fornece uma solução para o problema ContagemAproximada($\mathbb{M}$, k), em que k é uma variável aleatória cujo valor é quase certamente $\lg \lg n$.

2.1 Algoritmo de Morris

Para armazenarmos *exatamente* um valor entre 0 e n, são necessários registradores com $\Omega(\lg n)$ bits. Então, para que seja possível contar o número de elementos em um conjunto $\mathbb{M}$ com até n elementos usando menos bits, devemos abandonar a *exatidão* da contagem e buscar alternativas aceitáveis.

Morris sugeriu utilizar um contador X para armazenar o valor de $\lg n$ ao invés de n. Neste caso, o estimador *aproximado* $\hat{n}$ de n seria $2^X - 1$. Este "-1" na expressão faz com que essa aproximação se ajuste à situação em que $\mathbb{M} = \emptyset$. E um contador como este pode ser armazenado em $O(\lg X) = O(\lg \lg n)$ bits.

A política de incremento de X passa a ser a questão central do algoritmo. Morris propôs uma estratégia simples para realizar esse acréscimo: a cada novo elemento examinado de $\mathbb{M}$, o contador X seria incrementado com probabilidade 2^{-X}.

Essa estratégia pode ser vista no algoritmo Morris cuja implementação é baseada em uma ideia da seção *Algorithm* do artigo *Approximate counting algorithm* (Wikipedia, 2021). Essa versão expressa a condição de incremento do contador como um experimento de lançamento de moedas. Sempre que um novo elemento do conjunto $\mathbb{M}$ é examinado, o lançamento de X moedas é simulado. O contador é incrementado somente quando os resultados de todos os lançamentos forem cara. E a probabilidade deste evento ocorrer é 2^{-X}.

O algoritmo Morris a seguir recebe um conjunto $\mathbb{M} = \{e_1, e_2, \ldots, e_n\}$ e devolve um estimador $\hat{n}$ para n. Este estimador é da forma $2^X - 1$ em que X é um número inteiro não negativo, e veremos mais adiante que o seu valor esperado é n, ou seja, $\mathbb{E}\left(\hat{n}\right) = \mathbb{E}\left(2^X - 1\right) = n$.

```
Morris(M)
1  X ← 0
2  para cada e em M
3      r ← número de caras resultantes dos lançamentos de X moedas
4      se r = 0
5          X ← X + 1
6  devolva 2^X − 1
```

Na prática, para mimetizar o lançamento de moedas, utilizamos uma função g geradora de números aleatórios que recebe o valor de X e devolve um inteiro aleatório $r = g(X)$ entre 0 e $2^X - 1$. Este número r pode ser visto como um inteiro de X bits em que cada bit representa o resultado do lançamento de uma moeda: 0 corresponde à coroa e 1, à cara.

```
MORRIS(𝕄)
1  X ← 0
2  para cada e em 𝕄
3        r ← g(e)
4        se r = 0
5              X ← X + 1
6  devolva 2^X − 1
```

2.2 Qualidade da aproximação

Esta seção busca investigar a qualidade do estimador devolvido pelo algoritmo MORRIS para o número n de elementos em um dado conjunto $\mathbb{M}$. As demonstrações apresentadas aqui são baseadas nas notas de aula de (ANDONI, 2017).

Primeiro, considere que MORRIS($\mathbb{M}$) é o estimador devolvido pelo algoritmo tendo o conjunto $\mathbb{M}$ como entrada. Note que devido às chamadas feitas ao gerador de números aleatórios g na linha 3, MORRIS($\mathbb{M}$) é uma variável aleatória. Da mesma forma, a variável X criada na linha 1 é também uma variável aleatória. Esta variável é interessante para a demonstração, uma vez que, o algoritmo MORRIS mantém a seguinte relação invariante envolvendo X:

> **Relação invariante** de MORRIS: no início da linha 2, vale que $\mathbb{E}\left(2^X - 1\right) = k$, em que, k é o número de elementos de $\mathbb{M}$ já examinados.

Ao final do algoritmo, temos que $\mathbb{E}(\text{MORRIS}(\mathbb{M})) = \mathbb{E}\left(2^X - 1\right)$ e que o número de elementos de $\mathbb{M}$ examinados é n. Assim, o seguinte teorema segue como consequência dessa relação invariante:

Teorema 1 (do valor esperado de MORRIS($\mathbb{M}$))**.** *Se $\mathbb{M}$ é um conjunto com n elementos, então* $\mathbb{E}(\text{MORRIS}(\mathbb{M})) = n$.

Denotemos por X_k o valor da variável X na linha 2 após k elementos de $\mathbb{M}$ terem sido examinados no laço das linhas 2–5. Inspecionaremos o valor de $\mathbb{E}\left(2^{X_k} - 1\right)$ para alguns valores específicos de k antes de passarmos para a demonstração da relação invariante.

Quando nenhum elemento de $\mathbb{M}$ foi processado, o valor da variável X é igual ao seu valor inicial e dessa maneira, $X_0 = 0$. Segue que

$$\mathbb{E}\left(2^{X_0} - 1\right) = \mathbb{E}\left(2^0 - 1\right) = E(0) = 0\,.$$

Na primeira iteração do laço das linhas 2–5, temos que $r = g(0) = 0$ e consequente-

mente, $X_1 = 1$. Dessa forma,

$$\mathbb{E}\left(2^{X_1} - 1\right) = \mathbb{E}\left(2^1 - 1\right) = E(1) = 1\,.$$

A determinação de $\mathbb{E}(2^{X_2} - 1)$ é mais desafiadora. A partir deste ponto, é interessante explicitarmos a relação de recorrência geral de X_k:

$$X_k = \begin{cases} 0 & \text{se } k = 0 \\ X_{k-1} & \text{se } g(X_{k-1}) \neq 0 \\ X_{k-1} + 1 & \text{se } g(X_{k-1}) = 0\,. \end{cases} \tag{1}$$

A relação acima é decorrente das linhas 2–5. Na k-ésima iteração deste laço, a condicional da linha 4 garante que o valor de X_k será $X_{k-1} + 1$ se o número r gerado na linha 3 for zero. A probabilidade deste evento ocorrer é $\Pr(g(X_{k-1}) = 0) = 1 \,/\, 2^{X_{k-1}}$, uma vez que, r é um número aleatório escolhido uniformemente entre 0 e $2^{X_{k-1}} - 1$, inclusive. Nesse sentido, a probabilidade de o valor de X_k ser X_{k-1} é igual ao complementar de $\Pr(g(X_{k-1}) = 0)$ e assim,

$$\Pr(g(X_{k-1}) \neq 0) = 1 - \Pr(g(X_{k-1}) = 0) = 1 - 1 \,/\, 2^{X_{k-1}}\,.$$

Podemos, agora, obter a partir da recorrência (1), uma expressão *recursiva* para os valores de $2^{X_k} - 1$ para $k = 0, 1, 2, \ldots$.

$$2^{X_k} - 1 = \begin{cases} 0 & \text{se } k = 0 \\ 2^{X_{k-1}} - 1 & \text{se } g(X_{k-1}) \neq 0 \\ 2^{X_{k-1}+1} - 1 & \text{se } g(X_{k-1}) = 0\,. \end{cases} \tag{2}$$

E com a recorrência (2) e os valores de $\Pr(g(X_{k-1}) = 0)$ e $\Pr(g(X_{k-1}) \neq 0)$, podemos encontrar uma expressão para $\mathbb{E}\left(2^{X_k} - 1\right)$:

$$\begin{aligned}
\mathbb{E}\left(2^{X_k} - 1\right) &= \mathbb{E}\Bigg(\underbrace{\left(1 - \frac{1}{2^{X_{k-1}}}\right)}_{\Pr(\, g(X_{k-1}) \neq 0\,)} \times \left(2^{X_{k-1}} - 1\right) + \underbrace{\frac{1}{2^{X_{k-1}}}}_{\Pr(\, g(X_{k-1}) = 0\,)} \times \left(2^{X_{k-1}+1} - 1\right) \Bigg) \\
&= \mathbb{E}\left(2^{X_{k-1}} - 1 - \frac{2^{X_{k-1}}}{2^{X_{k-1}}} + \frac{1}{2^{X_{k-1}}} + \frac{2^{X_{k-1}+1}}{2^{X_{k-1}}} - \frac{1}{2^{X_{k-1}}} \right) \\
&= \mathbb{E}\left(2^{X_{k-1}} - 1 - 1 + \frac{1}{2^{X_{k-1}}} + 2 - \frac{1}{2^{X_{k-1}}} \right) \\
&= \mathbb{E}\left(2^{X_{k-1}}\right)\,.
\end{aligned} \tag{3}$$

E tomando $k = 2$ nesta expressão, obtemos

$$\mathbb{E}\left(2^{X_2} - 1\right) = \mathbb{E}\left(2^{X_{2-1}}\right) = \mathbb{E}\left(2^{X_1}\right) = \mathbb{E}\left(2^1\right) = 2\,.$$

Os cálculos envolvidos na determinação de $\mathbb{E}\left(2^{X_3} - 1\right)$ estão explicitados na árvore de

decisão da Figura 1. Nesta árvore, podemos encontrar os possíveis valores para as variáveis aleatórias X_1, X_2 e X_3, além das probabilidades de cada variável assumir um certo valor. As setas da cor verde indicam que o contador foi incrementado e as da cor vermelha, que o contador não sofreu mudanças. Assim, para que tenhamos, por exemplo, $X_3 = 3$, é necessário que durante as iterações do algoritmo, obtenhamos $g(1) = 0$ e $g(2) = 0$. Este evento ocorre com probabilidade $1 / 2 \times 1 / 4 = 1 / 8$. Na figura, este fato é indicado com a fração $1 / 8$ próxima à folha de rótulo $X_3 = 3$.

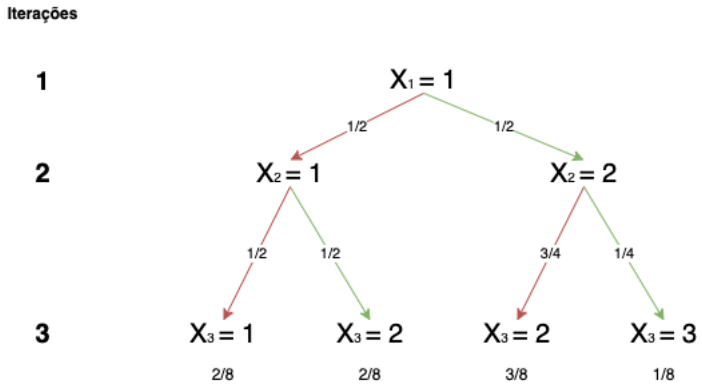


Figura 1: *Árvore de decisão para determinar* $\mathbb{E}\left(2^{X_3} - 1\right)$ *no algoritmo* Morris. *A fração próxima de cada folha indica a probabilidade de* X_3 *assumir o valor em seu rótulo.*

Inspecionando as folhas da árvore acima, vemos que

$$\begin{aligned}\mathbb{E}\left(2^{X_3} - 1\right) &= \frac{2}{8} \times \left(2^1 - 1\right) + \frac{2}{8} \times \left(2^2 - 1\right) + \frac{3}{8} \times \left(2^2 - 1\right) + \frac{1}{8} \times \left(2^3 - 1\right) \\ &= \frac{2}{8} \times 1 + \frac{2}{8} \times 3 + \frac{3}{8} \times 3 + \frac{1}{8} \times 7 \\ &= \frac{24}{8} \\ &= 3\,.\end{aligned}$$

Em todos os exemplos calculados até agora, vimos que $\mathbb{E}\left(2^k - 1\right) = k$. Após ganharmos intuição sobre $\mathbb{E}\left(2^k - 1\right)$ com valores pequenos de k, estamos preparados para demonstrar a validade da relação invariante.

Demonstração (da relação invariante). A prova será por indução no número k de elementos de $\mathbb{M}$ já examinados. Vimos anteriormente que $\mathbb{E}\left(2^0 - 1\right) = 0$ e assim, a relação vale para $k = 0$.

Suponhamos, agora, que a relação invariante vale para algum $k-1$, $k \geq 1$. Mostraremos que isso implica que esta relação vale para k. Dessa forma, temos que

$$\begin{aligned}\mathbb{E}\left(2^{X_k} - 1\right) &= \mathbb{E}\left(2^{X_{k-1}}\right) \\ &= \mathbb{E}\left(2^{X_{k-1}}\right) - 1 + 1\end{aligned} \tag{4}$$

$$
\begin{aligned}
&= \mathbb{E}\left(2^{X_{k-1}} - 1\right) + 1 \\
&= k - 1 + 1 \qquad (5) \\
&= k,
\end{aligned}
$$

em que a igualdade (4) é decorrência da identidade (3) e a igualdade (5), da hipótese de indução. Finalmente, a validade da relação invariante segue do princípio da indução finita. □

Saber que $\mathbb{E}(\textsc{Morris}(\mathbb{M})) = |\mathbb{M}| = n$ não é suficiente para garantir a qualidade do estimador $2^X - 1$ devolvido pelo algoritmo. É desejável que além do valor esperado ser próximo ou igual ao tamanho do conjunto $\mathbb{M}$, a variância da variável aleatória $\textsc{Morris}(\mathbb{M})$ não seja *muito grande*. Intuitivamente, isso nos diria que a probabilidade do estimador calculado ser longe do valor esperado é pequena. Então, apresentaremos o valor da variância de $\textsc{Morris}(\mathbb{M})$ no próximo teorema.

Teorema 2 (da variância)**.** *Se $\mathbb{M}$ é um conjunto com n elementos, então* $\mathbb{V}(\textsc{Morris}(\mathbb{M})) = n(n-1)/2$.

Demonstração (da variância). Pela definição da variância de uma variável aleatória, temos que

$$
\begin{aligned}
\mathbb{V}(\textsc{Morris}(\mathbb{M})) &= \mathbb{E}(\textsc{Morris}(\mathbb{M})^2) - \mathbb{E}^2(\textsc{Morris}(\mathbb{M})) \\
&= \mathbb{E}\left(\left(2^{X_n} - 1\right)^2\right) - n^2 \qquad (6) \\
&= \mathbb{E}\left(2^{2X_n} - 2\left(2^{X_n}\right) + 1\right) - n^2 \\
&= \mathbb{E}\left(2^{2X_n}\right) - 2\mathbb{E}\left(2^{X_n}\right) + 2 - 2 + 1 - n^2 \\
&= \mathbb{E}\left(2^{2X_n}\right) - 2\mathbb{E}\left(2^{X_n} - 1\right) - 2 + 1 - n^2 \\
&= \mathbb{E}\left(2^{2X_n}\right) - 2n - 1 - n^2\,, \qquad (7)
\end{aligned}
$$

em que utilizamos a identidade $\mathbb{E}(\textsc{Morris}(\mathbb{M})) = \mathbb{E}\left(2^{X_n} - 1\right) = n$ nas igualdades (6) e (7).

Logo em seguida, passaremos a calcular $\mathbb{E}\left(2^{2X_n}\right)$. Para isso, procederemos de maneira idêntica ao que foi feito na demonstração do Teorema do valor esperado utilizando a recorrência (1). Assim,

$$
\begin{aligned}
\mathbb{E}\left(2^{2X_n}\right) &= \mathbb{E}\left(\left(1 - \frac{1}{2^{X_n}}\right) \times 2^{2X_{n-1}} + \frac{1}{2^{X_n}} \times 2^{2(X_{n-1}+1)}\right) \\
&= \mathbb{E}\left(2^{2X_{n-1}} - \frac{2^{2X_{n-1}}}{2^{X_{n-1}}} + \frac{2^{2(X_{n-1}+1)}}{2^{X_{n-1}}}\right) \\
&= \mathbb{E}\left(2^{2X_{n-1}} - 2^{X_{n-1}} + 2^{X_{n-1}+2}\right) \\
&= \mathbb{E}\left(2^{2X_{n-1}} + 3 \times 2^{X_{n-1}}\right)
\end{aligned}
$$

$$\begin{aligned} &= \mathbb{E}\left(2^{2X_{n-1}}\right) + 3\mathbb{E}\left(2^{X_{n-1}}\right) \\ &= \mathbb{E}\left(2^{2X_{n-1}}\right) + 3n\,. \end{aligned} \tag{8}$$

A igualdade (8) segue da identidade (3). E o valor de $\mathbb{E}\left(2^{2X_n}\right)$ pode ser obtido da expansão *telescópica* da relação de recorrência

$$\begin{aligned} \mathbb{E}\left(2^{2X_n}\right) &= \mathbb{E}\left(2^{X_{n-1}}\right) + 3n \\ &= \mathbb{E}\left(2^{X_{n-2}}\right) + 3(n-1) + 3n \\ &= \mathbb{E}\left(2^{X_{n-3}}\right) + 3(n-2) + 3(n-1) + 3n \\ &\;\;\vdots \\ &= 1 + 3 + \cdots + 3(n-2) + 3(n-1) + 3n \\ &= \frac{3n(n+1)}{2} + 1\,. \end{aligned}$$

Finalmente, utilizando esse valor para prosseguir com (7), obtemos

$$\begin{aligned} \mathbb{V}(\textsc{Morris}(\mathbb{M})) &= \mathbb{E}\left(2^{2X_n}\right) - 2n - 1 - n^2 \\ &= \frac{3n(n+1)}{2} + 1 - 2n - 1 - n^2 \\ &= \frac{3n(n+1) - 4n - 2n^2}{2} \\ &= \frac{n(n-1)}{2}\,. \end{aligned}$$

□

Dessa forma, podemos estimar o erro do algoritmo Morris a partir dos Teoremas do valor esperado e da variância. Seja σ o desvio padrão de Morris(M). A desigualdade de Chebyshev nos diz que para todo $c > 0$,

$$\mathbb{P}\mathrm{r}(|\textsc{Morris}(\mathbb{M}) - n| \geq c\sigma) \leq \frac{1}{c^2}\,. \tag{9}$$

Pondo $c = 4\sqrt{2}$ em (9), obtemos que

$$\mathbb{P}\mathrm{r}\left(|\textsc{Morris}(\mathbb{M}) - n| \geq 4\sqrt{2}\times\sigma\right) \leq \frac{1}{(4\sqrt{2})^2} = \frac{1}{32}\,. \tag{10}$$

Como pelo Teorema da variância,

$$\sigma^2 = \mathbb{V}(\textsc{Morris}(\mathbb{M})) = \frac{n(n-1)}{2}\,,$$

vale que

$$\sigma < \frac{n}{\sqrt{2}} .$$

Logo, temos que

$$\Pr\left(|\textsc{Morris}(\mathbb{M}) - n| \geq 4\sqrt{2} \times \frac{2}{\sqrt{2}} \right) = \Pr(|\textsc{Morris}(\mathbb{M}) - n| \geq 4n) \leq \frac{1}{32} .$$

Portanto, podemos concluir que

$$\Pr(|\textsc{Morris}(\mathbb{M}) - n| \geq 4n) \leq \frac{1}{32} < 0{,}032 . \tag{11}$$

Em palavras, isto nos diz que com probabilidade maior que 96,8%, o erro relativo cometido por Morris(𝕄) é menor do que 4.

Por fim, voltemos nossa atenção para uma das motivações iniciais do trabalho de Morris: o espaço utilizado pelo algoritmo. Por espaço, queremos dizer o número de bits que o algoritmo Morris necessita para representar o contador X definido na linha 1. Denotaremos por $\textsc{bits}(i)$ como sendo o menor número de bits para representar um número inteiro não-negativo i. Temos que $\textsc{bits}(0) = 1$ e para $i \geq 1$, $\textsc{bits}(i) = \lfloor \lg i \rfloor + 1$.

Teorema 3. *Sejam* $\mathbb{M}$ *um conjunto com* $n \geq 2$ *elementos,* X *a variável definida na linha 1 do algoritmo* Morris *e* b *um número inteiro positivo. Se* X_n *é o valor da variável* X *ao final de* Morris(𝕄)*, então* $\Pr(\textsc{bits}(X_n) \geq b + \lg\lg n + 1) \leq 1/(n^{2^b - 1} - 1)$.

Consideremos um exemplo para decifrar o significado do Teorema 3. Suponhamos que $\mathbb{M}$ é um conjunto com $n = 2^{16} = 65536$ elementos. Neste caso, temos que $\lg\lg n = 4$. O teorema nos diz que para $b = 1$, a representação de $X = X_{65536}$ utilizará 6 bits com probabilidade menor ou igual a $1/(65536^{2^1 - 1}) = 1/65536 < 1{,}53 \times 10^{-5}$. E para $b = 2$, seriam necessários 7 bits com probabilidade menor ou igual a $1/(65536^{2^2 - 1}) < 3{,}56 \times 10^{-15}$. Nesse sentido, o Teorema 3 nos garante que o algoritmo Morris consome *quase certamente* não mais que $\lceil \lg\lg n \rceil$ bits de espaço.

Demonstração (do Teorema 3). A desigualdade de Markov nos diz que para todo número real positivo c vale que

$$\Pr(\textsc{Morris}(\mathbb{M}) \geq c) \leq \frac{\textsc{Morris}(\mathbb{M})}{c} = \frac{n}{c} . \tag{12}$$

Pondo $c = n^{2^b} - 1$ em (12), obtemos que

$$\Pr\left(\textsc{Morris}(\mathbb{M}) \geq n^{2^b} - 1\right) \leq \frac{1}{n^{2^b - 1} - 1} . \tag{13}$$

Como $\textsc{Morris}(\mathbb{M}) = 2^{X_n} - 1$, reescrevendo e desenvolvendo (13), encontramos que

$$\Pr\left(2^{X_n} - 1 \geq n^{2^b} - 1\right) = \Pr\left(2^{X_n} \geq n^{2^b}\right)$$

$$
\begin{aligned}
&= \Pr\left(X_n \geq \lg n^{2^b}\right) \\
&= \Pr\left(\lg X_n \geq \lg \lg n^{2^b}\right) \\
&\geq \Pr\left(\lfloor \lg X_n \rfloor \geq \lg \lg n^{2^b}\right) \\
&= \Pr\left(\lfloor \lg X_n \rfloor + 1 \geq \lg \lg n^{2^b} + 1\right) \\
&= \Pr\left(\textsc{bits}(X_n) \geq \lg \lg n^{2^b} + 1\right) \\
&= \Pr\left(\textsc{bits}(X_n) \geq \lg 2^b + \lg \lg n + 1\right) \\
&= \Pr(\textsc{bits}(X_n) \geq b + \lg \lg n + 1) \,.
\end{aligned}
$$

Combinando a desigualdade acima com (13), vemos que

$$
\Pr(\textsc{bits}(X_n) \geq b + \lg \lg n + 1) \leq \frac{1}{n^{2^{b-1}} - 1} \,.
$$

Com isto, concluímos a prova do teorema. □

2.3 Morris e a Lei dos Grandes Números

Do teorema da variância de Morris(𝕄), notamos que o algoritmo apresenta uma grande variabilidade. Nesta seção, veremos uma técnica que a partir de um algoritmo probabilístico com uma certa variância, obteremos um novo algoritmo com variância menor.

Em nosso caso, nos apoiaremos no algoritmo Morris para construir o algoritmo Morris++ que tem o mesmo valor esperado e menor variância que Morris. Como foi visto em (9), a desigualdade de Chebyshev limita o erro em função da variância, e assim, o erro relativo de Morris++ também será menor que o de Morris. E as demonstrações apresentadas aqui serão também baseadas nas notas de aula de (Andoni, 2017).

O algoritmo Morris++ recebe um conjunto $\mathbb{M} = \{e_1, e_2, \ldots, e_n\}$ e um número positivo t, e devolve um estimador $\hat{n}$ para n. Este estimador será a média aritmética de t execuções de Morris(𝕄). Esta estratégia é inspirada na Lei dos Grandes Números, que diz que a média dos resultados obtidos de um grande número de experimentos deve se aproximar do valor esperado conforme mais experimentos são realizados.

```
Morris++(𝕄, t)
1  N ← 0
2  para i ← 1 até t
3        n_i ← Morris(𝕄)
4        N ← N + n_i
5  devolva N / t
```

As variáveis $n_1, n_2, \ldots, n_t$ definidas na linha 3 são aleatórias e tais que

$$
\mathbb{E}(n_i) = \mathbb{E}(\textsc{Morris}(\mathbb{M})) = n \tag{14}
$$

$$
\mathbb{V}(n_i) = \mathbb{V}(\textsc{Morris}(\mathbb{M})) = n(n-1)/2 \,, \tag{15}
$$

para $i = 1, 2, \ldots, t$. A igualdade (14) é decorrência do teorema do valor esperado de Morris(𝕄), e a igualdade (15), do teorema da variância de Morris(𝕄).

De maneira semelhante ao que já havíamos feito, denotaremos por Morris++$(\mathbb{M}, t)$ como sendo o estimador devolvido pelo algoritmo tendo o conjunto 𝕄 e o número inteiro t como entrada. É evidente que Morris$(\mathbb{M})$ = Morris++$(\mathbb{M}, 1)$.

Teorema 4 (do valor esperado de Morris++)**.** *Se* 𝕄 *é um conjunto com* n *elementos e* t*, um número inteiro positivo, então* $\mathbb{E}(\text{Morris++}(\mathbb{M}, t)) = n$.

Demonstração. A variável aleatória N é definida na linha 1. Seja N_t o valor desta variável ao final da execução do algoritmo. Temos que

$$\begin{aligned}
\mathbb{E}\big(\text{Morris++}(\mathbb{M}, t)\big) &= \mathbb{E}\left(\frac{N_t}{t}\right) \\
&= \mathbb{E}\left(\frac{n_1 + n_2 + \cdots + n_t}{t}\right) \\
&= \frac{\mathbb{E}(n_1) + \mathbb{E}(n_2) + \cdots + \mathbb{E}(n_t)}{t} \\
&= \frac{\mathbb{E}(\text{Morris}(\mathbb{M})) + \cdots + \mathbb{E}(\text{Morris}(\mathbb{M}))}{t} \qquad (16) \\
&= \frac{1}{t} \times t \times \mathbb{E}(\text{Morris}(\mathbb{M})) \\
&= \mathbb{E}(\text{Morris}(\mathbb{M})) \\
&= n\,, \qquad (17)
\end{aligned}$$

em que as igualdades em (16) e (17) seguem das identidades em (14). □

Teorema 5 (da variância de Morris++)**.** *Se* 𝕄 *é um conjunto com* n *elementos e* t *é um número inteiro positivo, então* $\mathbb{V}(\text{Morris++}(\mathbb{M}, t)) = \mathbb{V}(\text{Morris}(\mathbb{M}))/t = n(n-1)/2t$.

Demonstração. Retome a definição de N_t na demonstração do valor esperado. Temos que

$$\begin{aligned}
\mathbb{V}\big(\text{Morris++}(\mathbb{M}, t)\big) &= \mathbb{V}\left(\frac{N_t}{t}\right) \\
&= \mathbb{V}\left(\frac{n_1 + n_2 + \cdots + n_t}{t}\right) \\
&= \frac{1}{t^2} \times \mathbb{V}(n_1 + n_2 + \cdots + n_t) \\
&= \frac{1}{t^2} \times \left(\sum_{i=1}^{t} \mathbb{V}(n_i) + 2 \times \sum_{i<j} \mathbb{C}\text{ov}(n_i, n_j)\right) \qquad (18) \\
&= \frac{1}{t^2} \times \sum_{i=1}^{t} \mathbb{V}(n_i) \qquad (19)
\end{aligned}$$

$$= \frac{1}{t^2} \times \sum_{i=1}^{t} \mathbb{V}(\text{Morris}(\mathbb{M})) \tag{20}$$

$$= \frac{1}{t^2} \times t \times \mathbb{V}(\text{Morris}(\mathbb{M}))$$

$$= \frac{1}{t} \times \frac{n(n-1)}{2} \tag{21}$$

$$= \frac{n(n-1)}{2t} .$$

Para todo $i < j$ as variáveis aleatórias n_i e n_j são independentes e portanto não correlacionadas, ou seja, $\mathbb{C}\text{ov}(n_i, n_j) = 0$. Isto implica a igualdade (19). As igualdades (20) e (21) são devidas às identidades em (15). □

Passemos a explorar como o parâmetro t de Morris++$(\mathbb{M}, t)$ pode nos ser útil para obtermos estimadores de interesse para o número de elementos em um dado conjunto $\mathbb{M}$ com n elementos.

Sejam ϵ e γ números reais tais que $0 < \gamma \leq 1$ e $\epsilon > 0$. Desejamos que

$$\mathbb{P}\text{r}(|\text{Morris++}(\mathbb{M}, t) - n| \geq \epsilon n) \leq \gamma .$$

Seja σ_t o desvio padrão de Morris++$(\mathbb{M}, t)$. Do teorema do valor esperado de Morris++ junto com a desigualdade de Chebyshev, obtemos que

$$\mathbb{P}\text{r}(|\text{Morris++}(\mathbb{M}, t) - n| \geq c\sigma_t) \leq \frac{1}{c^2} .$$

O teorema da variância de Morris++ nos diz que $\sigma_t^2 = \mathbb{V}(\text{Morris++}(\mathbb{M}, t)) = n(n-1)/2t$ e portanto, $\sigma_t < \frac{n}{\sqrt{2t}}$. Logo,

$$\mathbb{P}\text{r}\left(|\text{Morris++}(\mathbb{M}, t) - n| \geq c \times \frac{n}{\sqrt{2t}}\right) \leq \frac{1}{c^2} . \tag{22}$$

Pondo $c = \gamma^{-1/2}$ em (22), chegamos em

$$\mathbb{P}\text{r}\left(|\text{Morris++}(\mathbb{M}, t) - n| \geq \frac{n}{\sqrt{2\gamma t}}\right) \leq \frac{1}{(\gamma^{-1/2})^2} = \gamma . \tag{23}$$

Tomando, agora, $t = \lceil (2\gamma\epsilon^2)^{-1} \rceil$, podemos concluir que

$$\frac{n}{\sqrt{2\gamma t}} = \frac{n}{\sqrt{2\gamma \lceil (2\gamma\epsilon^2)^{-1} \rceil}} \leq \frac{n}{\sqrt{2\gamma (2\gamma\epsilon^2)^{-1}}} = \frac{n}{\sqrt{\epsilon^{-2}}} = \frac{n}{\epsilon^{-1}} = \epsilon n .$$

Ao combinarmos a desigualdade anterior com a desigualdade em (23), encontramos a

desigualdade desejada:

$$\Pr(|\textsc{Morris++}(\mathbb{M}, t) - n| \geq \epsilon n) \leq \gamma \ .$$

A discussão acima pode ser resumida no teorema a seguir:

Teorema 6 (do erro de Morris++)**.** *Sejam ϵ e γ números reais tais que $\epsilon > 0$ e $0 < \gamma \leq 1$. Se $\mathbb{M}$ é um conjunto com n elementos, então*

$$\Pr(|\textsc{Morris++}(\mathbb{M}, t) - n| \geq \epsilon n) \leq \gamma \ ,$$

em que $t = \lceil (2\gamma\epsilon^2)^{-1} \rceil$.

Vamos testar o Teorema (6) para ganharmos intuição sobre seu significado. Suponha que desejamos que

$$\Pr(|\textsc{Morris++}(\mathbb{M}, t) - n| \geq 4n) \leq 0{,}032 \ .$$

Ou seja, queremos que a probabilidade do erro relativo ser maior que 4 seja menor que 0,032. Em outras palavras, desejamos que o erro relativo seja menor que 4 em pelos menos 96,8% das vezes que executarmos Morris++$(\mathbb{M}, t)$. Como nesta situação, temos que $\epsilon = 4$ e $\gamma = 0{,}03$, o valor de t para garantir esse desempenho é dado por

$$t = \lceil (2\gamma\epsilon^2)^{-1} \rceil = \left\lceil \frac{1}{2 \times 0{,}032 \times 4^2} \right\rceil = \left\lceil \frac{1}{1024} \right\rceil = \lceil 0{,}9765625 \rceil = 1 \ .$$

Assim, para obtermos o desempenho acima, precisamos executar pelo menos uma vez a subrotina Morris dentro do algoritmo Morris++. Este resultado bate com os cálculos desenvolvidos na seção anterior que nos levaram até a desigualdade (11).

Vamos considerar outro exemplo em que desejamos que

$$\Pr(|\textsc{Morris++}(\mathbb{M}, t) - n| \geq 0{,}1n) \leq 0{,}05 \ .$$

Ou seja, gostaríamos que o erro fosse menor que 10% com probabilidade maior que 95%. Nesta nova situação, temos que $\epsilon = 0{,}1$ e $\gamma = 0{,}05$. E o valor de t para garantir esse desempenho é

$$t = \lceil (2\gamma\epsilon^2)^{-1} \rceil = \left\lceil \frac{1}{2 \times 0{,}05 \times 0,1^2} \right\rceil = \left\lceil \frac{1}{0{,}001} \right\rceil = \lceil 1000 \rceil = 1000 \ . \tag{24}$$

Portanto, o algoritmo Morris++ deve executar a subrotina Morris pelo menos 1000 vezes.

2.4 Um, dois, três, ...

Nesta seção, apresentaremos as implementações da **versão dinâmica** dos algoritmos Morris e Morris++, além de simulações dessas soluções com o objetivo de verificar

experimentalmente a acurácia dos cálculos das seções anteriores.

Primeiramente, vamos lembrar as diferenças entre uma solução **estática** e **dinâmica** para o problema da contagem aproximada: enquanto que na versão estática, passamos um conjunto $\mathbb{M}$ para o *algoritmo* que devolve uma estimativa para a quantidade de elementos em $\mathbb{M}$, na versão dinâmica, construiremos uma *estrutura de dados* que realizará as operações de adição e contagem de itens. Nesse sentido, consideraremos que $\mathbb{M}$ é um fluxo de dados $x_1, x_2, \ldots, x_n, \ldots$, e estaremos interessados em saber a estimativa da quantidade de elementos que já foram adicionados na estrutura em um dado instante.

Vamos, agora, elaborar uma estrutura de dados que resolve o problema da contagem aproximada tomando como base o algoritmo Morris. Como discutido anteriormente, essa estrutura precisa suportar as operações de adição e contagem de elementos. Dessa forma, as linhas 2–5 do algoritmo Morris constituirão a adição, enquanto que a linha 6, a contagem. Note que essas duas operações utilizam a variável X definida na linha 1.

Apresentaremos, a seguir, uma implementação na linguagem Python da estrutura de dados acima. Utilizaremos orientação a objetos para resolver esse problema e por isso, criaremos uma classe `Morris` com os métodos `adiciona` e `conta`. Essa classe possuirá a variável de instância X que será compartilhada pelos métodos citados.

```
1  class Morris:
2    def __init__(self):
3        self.X = 0
4
5    def adiciona(self):
6        r = randint(0, 2 ** self.X - 1)
7        if r == 0:
8            self.X = self.X + 1
9
10   def conta(self):
11       return 2 ** self.X - 1
```

Programa 2.1: *Implementação do algoritmo* Morris

Na implementação acima, a função *randint* da biblioteca random do python faz o papel do gerador g de números aleatórios do algoritmo Morris. Essa função recebe inteiros a e b, e retorna de modo uniforme um inteiro aleatório entre a e b inclusive. Dessa forma, no Programa 2.1, a probabilidade de a variável r receber o valor zero é $1 \,/\, 2^X$, que coincide com a chance do contador ser incrementado no algoritmo Morris.

Um comentário importante a se fazer sobre a implementação de geradores de números aleatórios, como a função *randint* é que na prática, funções deste tipo geram números **pseudo-aleatórios**. Isto quer dizer, na maioria das vezes, que o programa define um ponto de partida e salta para outros pontos de maneira previsível. E a aparente aleatoriedade deste processo é o desconhecimento do ponto inicial. Um exemplo simples de função geradora de números pseudo-aleatórios é o seguinte: tome três primos a, b, m. Defina um inteiro inicial s e outro inteiro t cujo valor é igual a s inicialmente. Dessa forma, toda vez que pedirmos para essa função gerar um número aleatório, retornaremos t e em seguida, igualaremos t a $(t \times b + a) \bmod m$, em que $x \bmod y$ é o resto da divisão de x por y.

Vamos ver um exemplo para deixar a ideia acima mais clara. Tomemos $a = 101$, $b = 10^9 + 7 = 1000000007$, $m = 10^9 + 9 = 1000000009$ e $s = 35769$. Inicialmente, temos que $t = 35769$. Logo, o primeiro número a ser gerado é 35769, e o valor de t passa a ser $(35769 \times 1000000007 + 101) \bmod 1000000009 = 999928572$. E o segundo número gerado é 999928572, e t passa a ser $(999928572 \times 1000000007 + 101) \bmod 1000000009 = 142975$.

Seja t_i o i-ésimo número gerado no processo descrito anteriormente. Dessa forma, $t_0 = 35769$, $t_1 = 999928572$ e $t_2 = 142975$. Geraremos mais alguns valores de t: $t_3 = 999714160$, $t_4 = 571799$, $t_5 = 998856512$, $t_6 = 2287095$ e $t_7 = 995425920$. Os primeiros números gerados usando essa técnica são, portanto: 35769, 999928572, 142975, 999714160, 571799, 998856512, 2287095, 995425920. E se passarmos essa lista para qualquer pessoa, ela provavelmente falaria que esses números foram gerados ao acaso.

A justificativa para pararmos para entender esses geradores de números pseudo-aleatórios é que as demonstrações de algoritmos probabilísticos utilizam geralmente funções que realmente geram números aleatórios. A prova do algoritmo Morris, por exemplo, supõe a existência da função g que gera uniformente inteiros menores que 2^X. Contudo, podemos usar um gerador de inteiros pseudo-aleatórios no lugar de g sem necessariamente perder as garantias do erro do algoritmo. As simulações a serem exibidas a seguir devem mostrar indícios que de fato, a precisão do algoritmo de Morris não fica comprometida com essa mudança.

O primeiro experimento a ser apresentado é simular uma situação em que um milhão de elementos são inseridos na estrutura `Morris`. Para fazermos isso, chamamos o método `adiciona` do Programa 2.1 um milhão de vezes. Para cada vez que executamos esse método, calculamos o erro relativo do algoritmo, ou seja, na i-ésima iteração da simulação, após o método `adiciona` ter sido chamado, chamamos o método `conta` e armazenamos seu retorno em uma variável temporária TEMP. Assim, o erro relativo naquela iteração foi $(\text{TEMP} - i) \,/\, i$, uma vez que, i pode ser visto como a quantidade de itens que já passaram pela estrutura.

Dessa forma, a Figura 2 exibe um gráfico cujo eixo x das abscissas representa as iterações da simulação e o eixo y das ordenadas, o retorno do método `conta`. A linha laranja do gráfico ilustra a evolução da **estimativa** de itens inseridos ao longo das iterações e a linha azul, evolução da quantidade **real** de itens inseridos ao longo das iterações. Assim, analisando essas duas linhas, podemos ter uma ideia se a estimativa do algoritmo se aproxima ou se distância do valor correto conforme mais iterações acontecem.

No entanto, somente com esse gráfico, pode ser difícil visualizar o erro relativo do algoritmo ao longo das iterações. É por isso que construímos também, o gráfico do erro relativo na Figura 2, em que o eixo x representa novamente as iterações da simulação e o eixo y, o erro relativo.

Para as primeiras cem iterações da simulação, é possível perceber que o erro relativo não é maior que 80%. Na seção sobre a qualidade da aproximação do algoritmo Morris, chegamos à conclusão que o erro relativo cometido por Morris($\mathbb{M}$) é quase certamente menor do que 4. Nesse sentido, o pedaço inicial da simulação está dentro do esperado. Vamos agora, verificar se esse resultado esperado se mantém conforme mais iterações são realizadas.

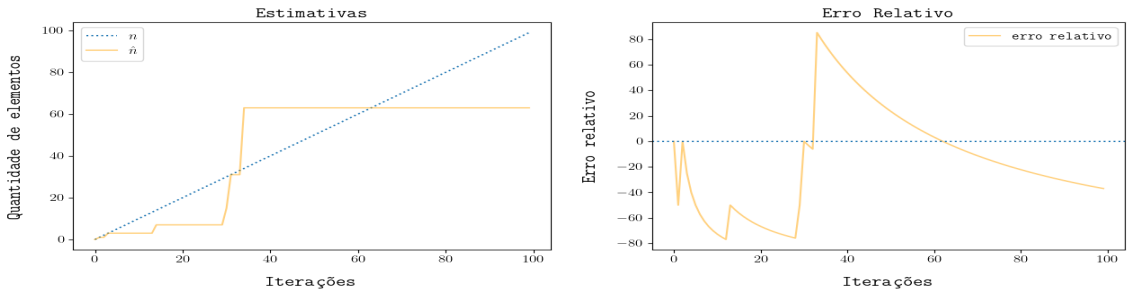


Figura 2: *Primeiras iterações do primeiro experimento do algoritmo* Morris. *Foram inseridos em uma estrutura* `Morris`*, um milhão de elementos. Em outras palavras, o método* `adiciona` *foi chamado um milhão de vezes. Depois de cada inserção, o resultado do método* `conta` *foi utilizado para a construção dos gráficos com a evolução da estimativa e do erro relativo.*

A Figura 3, que exibe um gráfico com erro relativo ao longo de toda a simulação, nos leva à conclusão que o erro relativo do algoritmo ficou dentro da faixa esperada, que é menos que 4. Essa mesma figura, que também exibe um gráfico que destaca a evolução da estimativa da quantidade de itens inseridos ao longo de toda a simulação, ressalta uma importante característica da estrutura: que quanto mais itens tiverem sido inseridos, mais demorado é para que a estimativa cresça. Isto está diretamente relacionado com a probabilidade do contador X ser incrementado no algoritmo Morris. Contudo, esse incremento demorado não é uma certeza absoluta, uma vez que, existem casos nessa simulação em que um crescimento da estimativa ocorreu em menos iterações que o crescimento anterior.

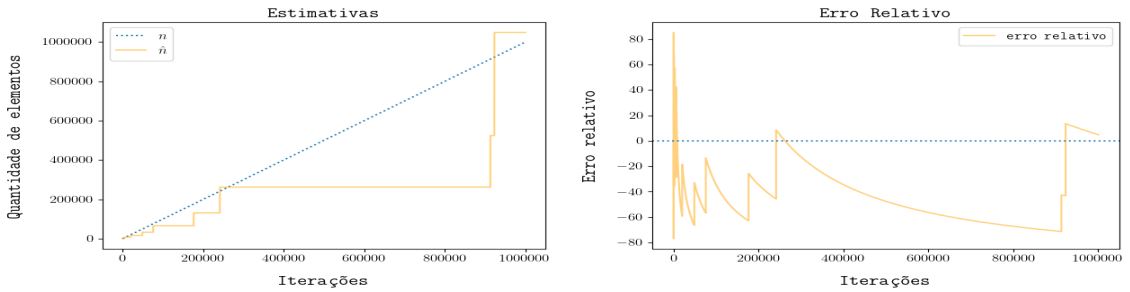


Figura 3: *Todas as iterações do primeiro experimento do algoritmo* Morris.

Uma dessas situações pode ser vista na Figura 4. Nessa figura, as linhas verticais indicam as iterações nas quais o contador foi incrementado. Assim, o contador foi incrementado nas iterações de números 75720, 176384 e 241343.

Para que houvesse o incremento na iteração 176384, precisaram ser feitas $176384 - 75720 = 100664$ iterações. Dessa forma, esperamos que o próximo incremento aconteça em pelo menos duzentas mil iterações. Só que ele ocorre antes, após $241343 - 176384 = 64959$ iterações.

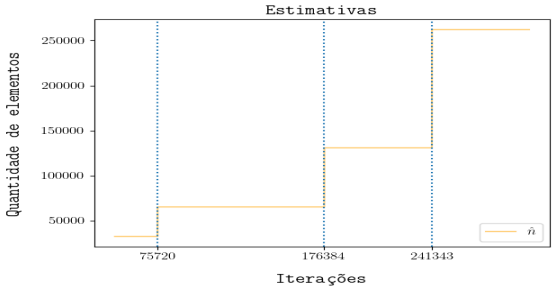


Figura 4: *Situação no primeiro experimento do algoritmo* Morris*, em que o contador foi incrementado mais cedo que o esperado. As linhas verticais indicam as iterações nas quais ocorreram um incremento do contador.*

O próximo experimento a ser apresentado consiste em repetir a simulação anterior várias vezes para que possamos ter uma ideia da variância da estrutura `Morris`. Dessa forma, repetimos dez mil vezes a simulação anterior e guardamos as frequências das estimativas devolvidas pelo algoritmo Morris. A distribuição dessas frequências pode ser vista na Figura 5, e para construir o eixo das abscissas, utilizamos o valor do contador `X` da classe `Morris` após todos os elementos terem sido inseridos. Nesse sentido, esse eixo apresentará em quantas simulações o contador `X` terminou por exemplo, com o valor 19. Nos experimentos realizados, isso aconteceu 3617 vezes.

Analisando a Figura 5, podemos concluir que na maioria dos casos, o erro relativo do algoritmo Morris ficou menor ou igual que 4, como foi previsto em (11). A melhor aproximação devolvida pelo algoritmo acontece quando $X = 20$, ou seja, a estimativa devolvida é $2^{20} - 1 \approx 1.000.000$. E assim, para até $X = 18$, estaríamos tendo um erro de quatro vezes menos, e $X = 22$, quatro vezes mais.

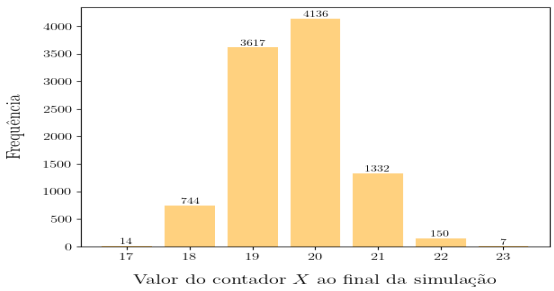


Figura 5: *Segundo experimento do algoritmo* Morris*. Foram realizadas dez mil simulações. Em cada simulação, foram inseridos um milhão de elementos em uma estrutura* `Morris` *e ao final de todas as inserções, o resultados do método* `conta` *foi utilizado para construir o gráfico com as distribuições das estimativas devolvidas pelo algoritmo.*

Com esses dois experimentos realizados, foi possível termos uma ideia da variância e precisão do algoritmo Morris. Dessa forma, podemos partir para a implementação da versão aprimorada desse algoritmo. Essa implementação terá como base Morris++. Como no algoritmo Morris++, utilizamos o algoritmo Morris como subrotina, vamos aproveitar o fato que temos a classe `Morris` já implementada e utilizá-la para implementar a classe `MorrisPlus`.

```python
class MorrisPlus:
  def __init__(self, t: int):
      self.t = t
      self.morris = [Morris() for _ in range(0, self.t)]

  def adiciona(self):
      for i in range(0, self.t):
          self.morris[i].adiciona()

  def conta(self):
      sum = 0
      for i in range(0, self.t):
          sum += self.morris[i].conta()

      return sum // self.t
```

Programa 2.2: *Implementação do algoritmo* Morris++

O parâmetro `t` da classe `MorrisPlus` indica quantas estruturas `Morris` estão sendo mantidas. Em outras palavras, esse parâmetro tem o mesmo significado da variável t no algoritmo Morris++, que é quantas vezes executaremos o algoritmo Morris para obtermos uma média das estimativas.

Vamos, agora, repetir os mesmos experimentos feitos para a estrutura `Morris` só que utilizando a implementação `MorrisPlus`. O esperado é que vejamos a Lei dos Grandes Números em ação e que a variância do Programa 2.2 seja menor. E o valor de t utilizado nos próximos experimentos foi de 1000 e assim, poderemos comparar os resultados deles com as contas desenvolvidas na seção sobre como aprimorar o algoritmo Morris.

O primeiro experimento a ser repetido é aquele em que desejamos observar a evolução do estimador devolvido pela estrutura `MorrisPlus`. Para isso, chamamos o método `adiciona` um milhão de vezes e construímos o gráfico com as estimativas ao longo das chamadas dessa função. O resultado dessa simulação pode ser visto no gráfico das estimativas da Figura 6, em que podemos notar que a linha laranja, que representa as estimativas devolvidas, se aproxima bastante da linha azul, que representa os valores esperados.

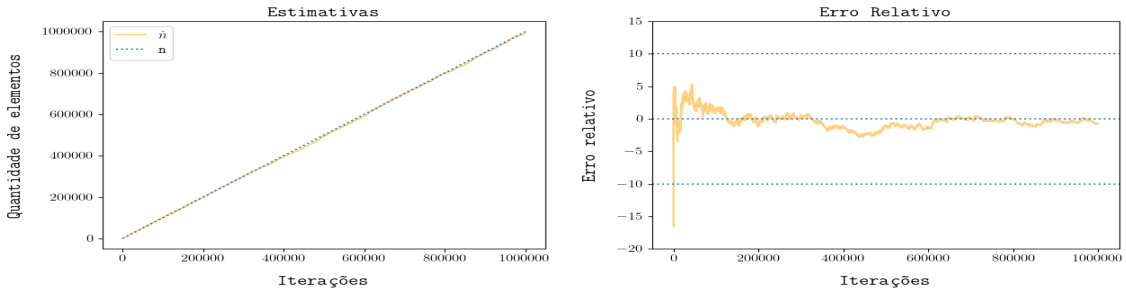


Figura 6: *Primeiro experimento do algoritmo* Morris++. *Foram inseridos em um estrutura* `MorrisPlus`*, um milhão de elementos.*

A Figura 6 também exibe um gráfico com a evolução do erro relativo cometido pela estrutura `MorrisPlus` ao longo das iterações da simulação anterior. Em (24), foi previsto

que se executássemos a subrotina Morris mil vezes dentro do algoritmo Morris++, deveríamos ter um erro relativo menor que 10% quase certamente. Contudo, no início da simulação, o erro relativo cometido foi maior que 15%. Isto é evidenciado na Figura 7, que destaca o erro relativo nas primeiras 50 iterações. Podemos, assim, desconsiderar essas iterações iniciais e ver como o erro evolui a partir da iteração de número 51. A Figura 7 também ilustra essa situação, e é possível perceber que o desvio cometido foi menor que 5%, que está dentro do previsto.

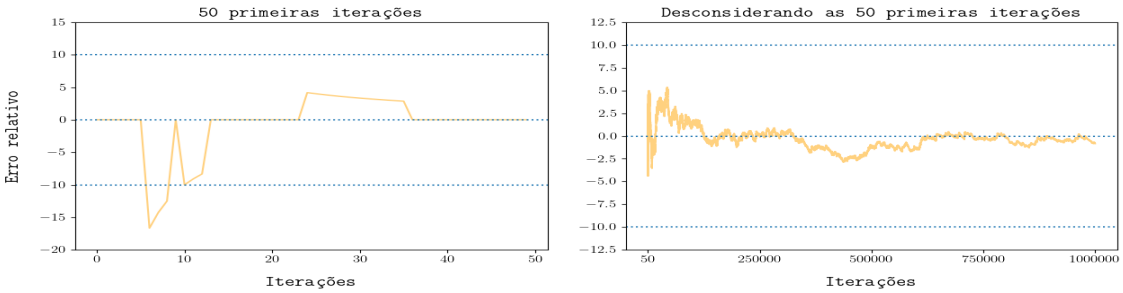


Figura 7: *Erro relativo do primeiro experimento do algoritmo* Morris++.

Podemos, agora, partir para o segundo experimento do algoritmo Morris++, cujo objetivo é verificar a variância dele. Para este experimento, repetimos a simulação anterior 100 vezes e coletamos as frequências dos estimadores devolvidos pela estrutura `MorrisPlus`. A partir desses dados, o histograma da Figura 8 foi construído. É possível, dessa forma, notar que as estimativas devolvidas ficaram dentro da faixa de erro relativo esperado, que é de 10%.

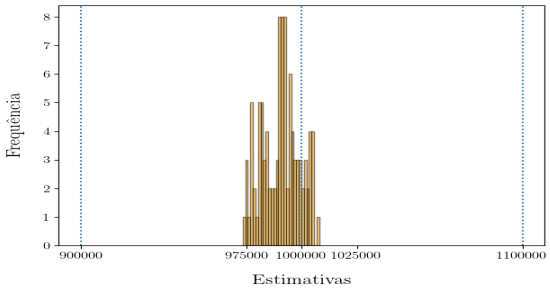


Figura 8: *Segundo experimento do algoritmo* Morris++. *Foram realizadas mil simulações e em cada simulação, foram inseridos um milhão de elementos em uma estrutura* Morris++. *Ao final de todas as inserções, as estimativas devolvidas foram usadas para construir o histograma de frequências.*

A partir dos experimentos apresentados, temos indícios que o algoritmo Morris++ tem uma precisão melhor que o algoritmo Morris e que mesmo este apresenta um desempenho razoável. Por fim, Morris conseguiu de fato resolver o problema de se manter contagens usando pouca memória, e veremos nas próximas seções, como a solução dele permitiu que problemas mais difíceis e relevantes fossem resolvidos.

Capítulo 3

Contagens Probabilísticas

Na seção anterior, abordamos a solução de Morris para o problema da contagem aproximada, que tinha como objetivo principal, estimar a quantidade de dados que passaram por um fluxo. Os fatos de, no contexto vivenciado por Morris, não ser possível armazenar todos os elementos nem manter a contagem em uma variável por conta de não existirem máquinas com registradores maiores que 8 bits tornam esse problema interessante.

Contudo, no contexto atual, em que os programas têm a disposição inteiros de 32 bits ou 64 bits, o problema da contagem aproximada parece não ser relevante. A razão disso é que o custo para mantermos um simples contador é praticamente nulo nos dias atuais, de modo que, uma solução probabilística não seria tão benéfica do ponto de vista do consumo de memória. Mesmo que esse problema não tenha aplicações imediatas atualmente, a solução proposta por Morris contém aspectos que inspiraram soluções de problemas mais desafiadores. Um desses problemas é descobrir o número aproximado de elementos distintos em um fluxo de dados $\mathbb{M}$.

> **Problema** da ContagemDistintaAproximada($\mathbb{M}$, k): Dado um fluxo de dados com repetições $\mathbb{M} = \{x_1, x_2, \dots, x_s, \dots\}$, **encontrar** estimador $\hat{n}$ para o número n de elementos *distintos* de $\mathbb{M}$ usando não mais que k bits.

Um modo de encontrarmos *exatamente* a quantidade de elementos distintos de $\mathbb{M}$ é inserir cada item novo que for aparecendo em um tabela e assim, o tamanho desta tabela será a resposta para o problema. A fragilidade dessa solução é o consumo de memória proporcional ao número de elementos distintos, e isto pode ser uma grande limitação em situações nas quais existam vários fluxos sendo monitorados.

Em muitos casos, não precisamos do valor *exato* da quantidade de elementos distintos em $\mathbb{M}$. Podemos assim, tentar projetar algoritmos *aproximados* e que consomem muito menos memória. Uma das primeiras soluções para o problema da contagem distinta aproximada foi elaborada por Philippe Flajolet e Nigel Martin (Flajolet e Martin, 1985). A motivação inicial desses autores era otimizar pesquisas em banco de dados relacionais, e a principal dificuldade nesse processo era calcular o número de elementos distintos em uma coluna. Nas próximas seções, veremos o algoritmo desenvolvido por Flajolet e Martin, conhecido como Contagem Probabilística.

3.1 Algoritmo da Contagem Probabilística

O algoritmo da Contagem Probabilística supõe a existência uma função de hash h que associa uniformemente cada elemento do fluxo $\mathbb{M}$ para um número inteiro entre 0 e $2^L - 1$ inclusive, em que L é a quantidade de bits que cada hash possui. O algoritmo também utiliza a função ρ que recebe um inteiro y de L bits e devolve a posição do primeiro dígito 1 da representação binária de y. Vamos ver exemplos dessa função para $L = 4$. Temos que $\rho(12) = \rho(0011_2) = 2$ e $\rho(9) = \rho(1001_2) = 0$. Um caso particular é quando $y = 0$, e nesta situação, $\rho(0) = L$. E teremos um vetor BMAP de tamanho $L + 1$ inicializado com zeros.

Assim, para cada x_i em $\mathbb{M}$, calcularemos um inteiro $y_i := h(x_i)$. Em seguida, encontraremos o valor de $\rho(y_i)$ e setaremos a posição $\rho(y_i)$ do vetor BMAP para 1. Dessa forma, esse vetor guardará quais valores $\rho(h(x_i))$ aparecem no fluxo $\mathbb{M}$.

Por fim, seja R o menor índice tal que $\text{BMAP}[R] = 0$. A estimativa para o número de elementos distintos de $\mathbb{M}$ será $2^R/\phi$, em que, ϕ é um fator de correção cujo valor é 0.77351... e cuja origem pode ser vista detalhadamente em (Flajolet e Martin, 1985).

Vamos ver um exemplo de execução do algoritmo descrito acima. Considere que $\mathbb{M} = \{4, 6, 8, 6, 9, 14\}$, $L = 4$ e que a função h é a identidade, ou seja, para todo x_i, temos que $h(x_i) = x_i$. Na primeira iteração, $x_1 = 4$ e $\rho(4) = \rho(0010_2) = 2$. Assim, setamos $\text{BMAP}[2] = 1$. Agora, na segunda iteração, $\rho(x_2) = \rho(6) = \rho(0110_2) = 1$, e setamos $\text{BMAP}[1] = 1$. Na terceira iteração, temos que $\rho(x_3) = \rho(8) = \rho(0001_2) = 3$ e portanto, $\text{BMAP}[3] = 1$. Na quarta iteração, o elemento 6 já apareceu, então BMAP permanece intacto. Na iteração seguinte, $\rho(x_6) = \rho(9) = \rho(1001_2) = 0$ e $\text{BMAP}[0] = 1$. Na última iteração, $\rho(x_7) = \rho(14) = \rho(0111_2) = 1$, mas a posição 1 de BMAP já está setada e dessa forma, BMAP permanece o mesmo. Falta, por fim, encontrar o valor de R. Como as posições 0, 1, 2, e 3 de BMAP estão preenchidas, temos que $R = 4$ e que a estimativa do número de elementos distintos de $\mathbb{M}$ é $2^R/\phi = 2^4/0.77351 \approx 21$. A estimativa anterior está muito distante do valor real, que é 5. Nas seções seguintes verificaremos se essa situação se repete para fluxos com muito mais elementos.

O algoritmo ProbabilisticCounting a seguir recebe um fluxo $\mathbb{M} = \{x_1, x_2, \ldots, x_s, \ldots\}$ com n elementos distintos, um inteiro L que representa quantos bits cada hash possui e uma função de hash h que mapeia os elementos $\mathbb{M}$ para inteiros entre 0 e $2^L - 1$ inclusive. E esse algoritmo devolve estimador $\hat{n}$ para n da forma $2^R/\phi$, em que, R é um número inteiro e ϕ, uma constante teórica. Veremos mais adiante que $\mathbb{E}(R) = \lg \phi n$, $\phi = 0.77351...$ e $\sigma(R) = 1.12$.

```
ProbabilisticCounting(𝕄, h, L)
1  para i de 0 até L + 1
2      BMAP[i] ← 0
3  para cada x em 𝕄
4      y ← h(x)
5      BMAP[ρ(y)] ← 1
6  R ← min{0 ≤ i ≤ L : BMAP[i] = 0}
7  devolva 2^R / φ
```

O consumo de espaço do algoritmo ProbabilisticCounting depende principalmente do vetor BMAP. Os valores que podemos encontrar neste vetor são zero, quando um dado valor de ρ ainda não tiver aparecido, ou um, caso contrário. Nesse sentido, é possível gastarmos somente um bit para guardar essa informação. Esse algoritmo, portanto, tem um consumo de memória de pelo menos $O(L)$ bits.

3.2 Padrões nos bits

Na seção anterior, descrevemos o algoritmo da CONTAGEM PROBABILÍSTICA. Vamos, agora, tentar entender a intuição por trás dessa solução.

No algoritmo ProbabilisticCounting($\mathbb{M}, h, L$), a função h que mapeia uniformemente os elementos do fluxo $\mathbb{M}$ para inteiros entre 0 e $2^L - 1$ inclusive nos permite supor que o fluxo $\mathbb{M}$ é na verdade um fluxo de números inteiros no intervalo $[0, 2^L - 1]$. Nesse sentido, podemos considerar um fluxo $\mathbb{M}_h = \{h(x_1), h(x_2), \dots, h(x_s), \dots\} = \{y_1, y_2, \dots, y_s, \dots\}$, em que $x_1, x_2, \dots$ são elementos de $\mathbb{M}$.

Dessa forma, um elemento y_i de $\mathbb{M}_h$ pode ser visto como uma palavra binária aleatória de L bits em que cada bit é gerado independentemente com probabilidade $1/2$. Com esta interpretação, a função ρ agrupa essas palavras de L bits de acordo com seus prefixos. Assim, as palavras z_0 tais que $\rho(z_0) = 0$ são aquelas cujo primeiro dígito é 1, ou seja, tem prefixo da forma $0^0 1$. Já as palavras z_1 tais que $\rho(z_1) = 1$ são aquelas cujo bit ligado menos significativo está na segunda posição, ou em outros termos, tem prefixo da forma $0^1 1$. As palavras aleatórias de L bits podem, portanto, ser divididas em grupos de acordo com prefixos da forma 0^*1.

Se os bits de uma palavra aleatória são gerados independentemente e uniformemente, então para $R \geq 0$, a probabilidade de uma palavra ter um prefixo da forma $0^R 1$ é $1/2^{R+1}$. Probabilidade e Contagem são áreas da Matemática muito relacionadas, de modo que ao invés de nos perguntarmos qual a probabilidade de uma palavra ter um prefixo da forma $0^R 1$, podemos nos questionar quantas palavras tem esse prefixo. Neste caso, $1/2^{R+1}$ das palavras terão esse prefixo.

Assim, para um fluxo $\mathbb{M}_h$ com n elementos distintos, é possível se perguntar quantos destes elementos começam com o dígito 1. A resposta para esta pergunta é que **esperamos** que $n / 2$ palavras desse fluxo se iniciem com o dígito 1. E $n / 2^2 = n / 4$ palavras devem ter o prefixo $0^{2-1}1 = 01$. Da mesma maneira, $n / 2^3 = n / 8$ palavras devem possuir um prefixo da forma $0^{3-1}1 = 001$. O vetor BMAP, desse modo, armazena quais prefixos já apareceram no fluxo $\mathbb{M}_h$. Falta, agora, entender o papel da variável R.

Suponha que $R = 4$, ou seja, BMAP$[R] = 0$ e para $0 \leq i < R$, BMAP$[i] = 1$. Vamos usar a ideia vista acima para tentar estimar o número de elementos distintos que devem ter aparecido. Esperamos que $1 / 2$ dos elementos de $\mathbb{M}_h$ se iniciem com o dígito 1, logo, a cada dois elementos sorteados ao acaso de $\mathbb{M}_h$, pelo menos um deve começar com 1. Em outras palavras, devem ter aparecido pelo menos duas palavras para que BMAP$[0] = 1$. Vamos ver se esse raciocínio faz sentido para BMAP$[1] = 1$. O esperado é que $1 / 4$ dos elementos de $\mathbb{M}_h$ comecem com 01, que a cada quatro palavras sorteadas de $\mathbb{M}_h$, pelo menos uma comece com 01 e assim, devem ter aparecido quatro palavras para que BMAP$[1] = 1$. Analogamente,

devem ter aparecido pelo menos 8 palavras para que BMAP$[2] = 1$ e 16 palavras para que BMAP$[3] = 1$. Como BMAP$[R = 4] = 0$, então ainda não devem ter aparecido 32 palavras no fluxo $\mathbb{M}_h$. Portanto, a melhor estimativa nesse caso é afirmar que $\mathbb{M}_h$ possui $2^R = 16$ elementos distintos.

A ideia descrita nessa seção é a base para os algoritmos que resolvem o problema da contagem distinta aproximada, e entendê-la ajudará a vermos a origem das soluções. Na próxima seção, serão expostas alguns aspectos da prova do funcionamento do algoritmo da CONTAGEM PROBABILÍSTICA.

3.3 Qualidade da aproximação

Essa seção destacará as principais **etapas** da análise da CONTAGEM PROBABILÍSTICA feita por Flajolet e Martin (FLAJOLET e MARTIN, 1985). Dessa forma, os detalhes técnicos de alguns passos da demonstração podem ser explorados com mais detalhes a partir da leitura do artigo original. Suponha que a quantidade de elementos distintos de um fluxo $\mathbb{M}$ seja n. O valor esperado da variável R definida na linha 6 do algoritmo ProbabilisticCounting é aproximadamente $\lg \phi n$, em que $\phi = 0.77351...$ E o desvio padrão de R é em torno de 1.12.

O primeiro passo da análise é definir a variável aleatória R_n como sendo o valor da variável R ao final da execução do algoritmo ProbabilisticCounting para uma entrada $\mathbb{M}$ com n elementos distintos, função de hash h e inteiro L. Assim, o interesse principal da demonstração é encontrar fórmulas ou estimativas para:

- $p_{n,k} = \mathbb{P}(R_n = k)$: probabilidade de uma saída de ProbabilisticCounting ser igual a k
- $q_{n,k} = \mathbb{P}(R_n \geq k)$: probabilidade de uma saída de ProbabilisticCounting ser maior ou igual a k
- $\mathbb{E}(R_n)$: valor esperado de R_n
- $\mathbb{V}(R_n)$: variância de R_n

O primeiro teorema de (FLAJOLET e MARTIN, 1985) mostra uma fórmula *exata* e *discreta* para $q_{n,k}$. A principal ideia para encontrar essa fórmula é agrupar as palavras binárias por prefixos da forma 0^k1. Nesse sentido, define-se $E_k = \{x \mid \rho(x) = k\}$, ou seja, E_k é o conjunto de todas as palavras aleatórias com prefixos iguais a 0^k1. Da mesma forma, define-se $K_k = \{x \mid \rho(x) \geq k\}$. Em seguida, as diferentes entradas $\mathbb{M}$ com n elementos distintos são representadas por um polinômio:

$$P_k^{(n)} = (E_0 + E_1 + .. + E_{k-1} + K_k)^n.$$

O próximo passo é tentar expandir esse polinômio usando *inclusão e exclusão*, e associar uma medida de probabilidade para $E_0, E_1, .., E_{k-1}, K_k$. E a prova deste teorema termina encontrando uma relação entre $q_{n,k}$ e esta expansão de polinômio.

Em seguida, o Teorema 2 de (FLAJOLET e MARTIN, 1985) apresenta aproximações de $q_{n,k}$ para diferentes intervalos de k. E a consequência deste teorema é que conforme n cresce,

$q_{n,k}$ pode ser expresso por uma fórmula *aproximada* e *contínua*:

$$q_{n,k} \approx \psi(\frac{n}{2^k})$$

em que, $\psi(x) = \prod_{j\geq 0}(1 - e^{-x2^j})$.

Note que por definição, $p_{n,k} = q_{n,k} - q_{n,k+1}$. Assim, pode-se aproximar $p_{n,k}$:

$$p_{n,k} \approx \psi(\frac{n}{2^k}) - \psi(\frac{n}{2^{k+1}}) \, .$$

O interesse passa a ser, portanto, estimar $\mathbb{E}(R_n)$ a partir dessa fórmula aproximada de $p_{n,k}$, de maneira que

$$\mathbb{E}(R_n) = \sum_{k\geq 1} k p_{n,k} \approx \sum_{k\geq 1} k\left[\psi\left(\frac{n}{2^k}\right) - \psi\left(\frac{n}{2^{k+1}}\right)\right] .$$

Desse modo, define-se a função real $F(x)$ como sendo

$$F(x) = \sum_{k\geq 1} k\left[\psi\left(\frac{n}{2^k}\right) - \psi\left(\frac{n}{2^{k+1}}\right)\right] .$$

E o Lema 1 do artigo, afirma que

$$\mathbb{E}(R_n) = F(x) + O\left(\frac{1}{n^{0.49}}\right) ,$$

ou seja, que o valor esperado de R_n se aproxima de $F(x)$ conforme n cresce.

Em seguida, o Lema 2 apresenta o resultado da transformada de Mellin de $F(X)$. A principal razão para se calcular essa transformação é que podemos expressar a fórmula inversa da transformada de Mellin, que é $\mathbb{E}(R_n)$, como uma expansão assintótica cujos termos são resíduos dessa transformada. Assim, o Teorema 3*A* utiliza os Lemas 1 e 2, e o Teorema dos Resíduos para afirmar que

$$\mathbb{E}(R_n) = \lg \phi n + P(\lg n) + o(1) \, ,$$

em que, $P(x)$ é a expansão assintótica de $F(x)$ e $\phi = 0.77351...$, concluindo a prova que $\mathbb{E}(R_n) \approx \lg \phi n$.

A prova para a estimativa de $\mathbb{V}(R_n)$ segue os mesmos passos da prova anterior. Pela definição de variância, precisamos estimar $\mathbb{E}(R_n^2)$. Assim,

$$\mathbb{E}(R_n^2) = \sum_{k=1} k^2 p_{n,k} \approx G(x) \, ,$$

em que

$$G(x) = \sum_{k=1} k^2 p_{n,k} \, .$$

Dessa forma, encontramos a transformada de Mellin de $G(x)$ e analisamos a inversa desta transformação para estimar $\mathbb{E}(R_n^2)$.

3.4 Lei dos Grandes Números **novamente**

Na seção anterior, foi visto que $\mathbb{E}(R_n) \approx \lg \phi n$, em que $\phi \approx 0.77351$. Assim, se o valor do contador R no final do algoritmo ProbabilisticCounting para uma entrada com n elementos distintos for aproximadamente igual a $\lg \phi n$, então a saída desse algoritmo é aproximadamente n. A Tabela 3.1 mostra que, em alguns casos, a saída do programa é praticamente igual a n quando $R_n \approx \lg \phi n$.

n	$2^{\lg(\phi n)}/\phi$
50	49.99
500	500.0
5000	4999.99
50000	50000.0

Tabela 3.1: *Comparação entre n e a saída do algoritmo ProbabilisticCounting quando* $R_n \approx \lg \phi n$.

Contudo, $\sigma(R_n) \approx 1.12$, ou seja, o valor de R_n pode ser uma unidade maior ou menor que $\lg \phi n$, o que implica que a estimativa de n possa ser duas vezes maior ou menor que n. Logo, o algoritmo ProbabilisticCounting apresenta uma grande variabilidade.

De modo similar ao que foi visto em Morris e a Lei dos Grandes Números, podemos obter a partir do algoritmo ProbabilisticCounting, uma solução com variância menor. Assim, o algoritmo ProbabilisticCounting+ recebe um fluxo de dados $\mathbb{M}$, inteiros m e L, além de uma lista de m funções de hash $\mathbb{H}$, em que cada função $\mathbb{H}_i$ desse grupo mapeia uniformemente os elementos de $\mathbb{M}$ para o intervalo $[0, 2^L - 1]$. Esse algoritmo modificado manterá m vetores BMAP, de maneira que para cada elemento x de $\mathbb{M}$, para todo $0 \leq j < m$, vamos setar $\text{BMAP}_j[\rho(\mathbb{H}_j(x))] = 1$. Quando todos os itens de $\mathbb{M}$ forem percorridos, calcularemos $R_j = \{\min 0 \leq i \leq L : \text{BMAP}_j[i] = 0\}$ para todo $0 \leq j < m$ e em seguida, encontraremos a média desses valores obtendo $\overline{R}$. Por fim, a estimativa para o número de elementos distintos em $\mathbb{M}$ será $2^{\overline{R}} \,/\, \phi$.

Uma outra forma de vermos essa solução é que estamos executando em paralelo m funções ProbabilisticCounting e para isso, temos que passar uma função de hash diferente para cada execução, a fim de produzir vetores BMAP diferentes ao final.

ProbabilisticCounting+($\mathbb{M}, \mathbb{H}, L, m$)

```
1  para j de 0 até m
2      para i de 0 até L + 1
3          BMAP_j[i] ← 0
4  para j de 0 até m
5      para cada x em M
6          y ← H_j(x)
7          BMAP_j[ρ(y)] ← 1
8  para i de 0 até m
9      R[j] ← min{0 ≤ i ≤ L : BMAP_j[i] = 0}
10 R̄ ← Σ_{j=0}^{m-1} R[j] / m
11 devolva 2^R̄ / φ
```

Seja $\overline{R_n}$ o valor da variável $\overline{R}$ ao final da execução do algoritmo ProbabilisticCounting+ para uma entrada com n elementos distintos. De forma análoga aos resultados vistos em Morris e a Lei dos Grandes Números, podemos concluir que:

$$\mathbb{E}(\overline{R_n}) \approx \lg \phi n \quad \text{e} \quad \sigma(\overline{R_n}) \approx \frac{1.12}{\sqrt{m}} \, .$$

No entanto, o fato de precisarmos de uma função de hash distinta para cada iteração torna a solução acima inviável, uma vez que, encontrar funções que mapeiem na prática os elementos de um fluxo $\mathbb{M}$ de maneira uniforme não é uma tarefa simples. Para contornar esse problema, os autores propuseram o uso da **média estocástica**.

Essa ideia consiste em dividir os elementos da entrada em m lotes e usar parte da informação do hash dos elementos para definir em qual lote um elemento deve ir. As linhas 4-7 do algoritmo ProbabilisticCounting++ a seguir mostram como essa divisão é feita.

ProbabilisticCounting++(M,h,L)

```
1  para j de 0 até m
2      para i de 0 até L + 1
3          BMAP_j[i] ← 0
4  para cada x em M
5      lote ← h(x) mod k
6      y ← ⌊h(x) / k⌋
7      BMAP_lote[y] ← 1
8  para i de 0 até m
9      R[j] ← min{0 ≤ i ≤ L : BMAP_j[i] = 0}
10 R̄ ← Σ_{j=0}^{m-1} R[j] / m
11 devolva m × 2^R̄ / φ
```

Se a função de hash h distribuir os n elementos distintos do fluxo $\mathbb{M}$ uniformemente entre os m lotes, então esperamos que cada lote tenha aproximadamente $\frac{n}{m}$ elementos. Nesse caso, $2^{\overline{R_n}} / \phi$ seria uma aproximação para $\frac{n}{m}$. É devido a este fato que a saída do algoritmo ProbabilisticCounting++ é $\mathbf{m} \times 2^{\overline{R_n}} / \phi$, pois este valor é uma estimativa para

$\mathbf{m} \times \frac{n}{m} = n$.

Na Seção 3.1, vimos que no algoritmo ProbabilisticCounting, precisamos de pelo menos $O(L)$ bits para armazenarmos o vetor BMAP. Como o algoritmo ProbabilisticCounting++ tem m vetores BMAP, o custo de espaço dele é de pelo menos $O(mL)$ bits.

Por fim, o desvio padrão do algoritmo ProbabilisticCounting++ é em torno de $0{,}78/\sqrt{m}$. Isto quer dizer que para $m = 64$, o desvio esperado é em torno de 10%. Dessa forma, a escolha de m depende das exigências do problema. Se precisarmos de uma precisão maior, escolheremos um valor de m maior, mas teremos um algoritmo mais custoso em termos de tempo e espaço. Por outro lado, se a exigência é, por exemplo, decidir qual de dois conjuntos tem a menor quantidade de elementos distintos, podemos escolher um valor menor de m para fazer essa comparação, mesmo tendo um risco maior de escolher o conjunto errado.

3.5 Estimando valores pequenos com a Contagem Probabilística

Uma das principais razões para escolhermos algoritmos *probabilísticos* ao invés de algoritmos *exatos* é o **consumo de memória reduzido**. Como já foi abordado, para resolver o problema da contagem distinta de forma *exata*, podemos manter uma tabela de hash e conforme os elementos forem processados, adicionar a esta tabela somente aqueles elementos que não apareceram ainda. Assim, a quantidade de elementos distintos processados será o tamanho da tabela. Suponha que cada hash seja um inteiro de 4 bytes (32 bits) e que existam um milhão de itens distintos percorridos. Para armazenar essa tabela, gastaríamos pelo menos 4 MB. Por outro lado, se usarmos o algoritmo da Contagem Probabilística para resolver esse problema, o consumo de memória seria proporcional ao valor m e não dependeria da quantidade de itens distintos. Suponha que BMAP seja um vetor de bits e que tenhamos m vetores deste tipo. Se m for igual a 1024, então a memória consumida por esse algoritmo seria em torno de 4 KB, que é um consumo de espaço **1000 vezes menor** que a solução exata.

O desvio padrão da Contagem Probabilística para $m = 1024$ é em torno de 2,5%. Logo, para um fluxo de milhões de itens distintos, o erro esperado seria em torno das dezenas ou centenas de milhares. Com esse tipo de desvio, teríamos como mensurar a ordem de grandeza do número de itens distintos que já passaram por um fluxo com bastante confiança. Contudo, para utilizar a Contagem Probabilística, o viés desse algoritmo deve ser entendido.

O **viés** de um estimador é a diferença entre seu valor esperado e o valor real do parâmetro estimado. Nesse sentido, a estimativa da quantidade de elementos em um conjunto devolvida pelo algoritmo de Morris é *não-viesada*, uma vez que, o Teorema do valor esperado de Morris afirma que o valor esperado deste estimador é igual ao tamanho do conjunto estimado.

Na prova descrita em (Flajolet e Martin, 1985), o estimador de $\lg n$ é R_n, que representa o valor da variável R ao final da execução de ProbabilisticCounting, tendo como entrada um fluxo de dados com n elementos distintos. Vimos que o valor esperado de

R_n é aproximadamente $\lg \phi n$, que é diferente de $\lg n$. Portanto, R_n é um estimador *viesado* de $\lg n$. E analogamente, o estimador devolvido por ProbabilisticCounting++ também é *viesado*.

Então, além do erro padrão, existe também o viés do algoritmo da Contagem Probabilística, cujo valor é $1 + 0{,}31 \,/\, m$. Contudo, para valores de m maiores que 32, o viés se torna desprezível se comparado ao desvio padrão. Dessa forma, uma pessoa pode ser induzida a concluir que a escolha de m é o único cuidado que ela deve tomar ao utilizar esse algoritmo. No entanto, ainda existe o problema de estimar **baixas cardinalidades** com essa estrutura.

Suponha, portanto, que $m = 64$ e que o hash do único elemento que já passou por um fluxo $\mathbb{M}$ comece com 1. Logo, se passarmos $\mathbb{M}$ para ProbabilisticCounting++, existiria um vetor bmap cuja primeira posição teria o valor 1, e o restante dos vetores bmap teriam todos valores zero. Assim, $\overline{R}$ seria igual a $1 \,/\, 64$, e o estimador de n teria o valor de $64 \times 2^{\frac{1}{64}} \,/\, \phi \approx 83.64$. Esta estimativa está muito distante do valor real, que é $n = 1$.

Em vista disso, um tratamento especial deve ser dado para situações em que a cardinalidade do número de elementos distintos de um conjunto for baixa. Poderíamos, por exemplo, manter os elementos percorridos em uma tabela de hash, e quando essa tabela atingisse um certo tamanho, passaríamos a usar a estratégia original da Contagem Probabilística, que é preencher os vetores bmap.

O artigo (Flajolet e Martin, 1985) nomeia esse problema como **não-linearidade inicial**, e os autores afirmam que a estimativa se aproxima do valor esperado do estimador assim que a quantidade de elementos distintos atinge pelo menos os valores $10m{-}20m$. Assim, para $m = 64$, é esperado que essa imprecisão inicial desapareça quando aproximadamente $600{-}1200$ dados distintos passarem pelo fluxo.

Vale lembrar, no entanto, que a proposta inicial para a adoção de estruturas de dados probabilísticas é o alto consumo de memória conforme o volume dos dados cresce. Nesse sentido, esse problema da não-linearidade inicial pode ser desconsiderado dependendo da situação.

3.6 Implementando Contagem Probabilística

Nesta seção, apresentaremos a implementação da Contagem Probabilística. Além disso, experimentos similares ao da Seção 2.4 foram realizados com o objetivo de se verificar a acurácia dessa solução.

A implementação foi baseada no algoritmo ProbabilisticCounting++ com algumas modificações. Dessa forma, definimos uma classe `ProbabilisticCounting`, cujo construtor recebe os parâmetros m e L. E nesta classe, existem os métodos `adiciona` e `conta`. O primeiro adiciona um elemento à estrutura, e o segundo devolve a estimativa da quantidade de elementos distintos. Assim, as simulações se baseiam em uma versão *online* da Contagem Probabilística para que possamos observar como a estimativa evolui conforme novos itens são inseridos nessa estrutura.

```
1  class ProbabilisticCounting:
2      def __init__(self, m=64, L=64):
3          self.m: int = m
4          self.L: int = L
5          self.phi = 0.77351
6          self.BITMAP: List[List[int]] = []
7
8          for _ in range(0, self.m):
9              self.BITMAP.append([None] * self.L)
10
11         self.R: List[int] = [0] * self.m
12         self.R_sum = 0
13
14     def p(self, x: int):
15         return (x & -x).bit_length() - 1
16
17     def adiciona(self, x: int):
18         lote = x % self.m
19         y = x // self.m
20         self.BITMAP[lote][self.p(y)] = 1
21
22         while self.BITMAP[lote][self.R[lote]] == 1:
23             self.R[lote] += 1
24             self.R_sum += 1
25
26     def conta(self):
27         R = self.R_sum / self.m
28
29         return floor((self.m * pow(2, R)) / self.phi)
```

Programa 3.1: *Implementação do algoritmo ProbabilisticCounting++*

Vamos comentar alguns detalhes de implementação do Programa 3.1. O primeiro deles é relacionado ao método `adiciona`, que foi baseados nas linhas 4–9 do algoritmo ProbabilisticCounting++. O principal diferencial da implementação desse método na classe `ProbabilisticCounting` é que já mantemos o menor valor de R de cada lote atualizado, como pode ser visto no `while`. Dessa forma, a complexidade de se adicionar um elemento nessa estrutura é $O(1)$ amortizado.

O próximo comentário é relacionado ao método `p` da classe `ProbabilisticCounting`, que é a implementação da função ρ do algoritmo ProbabilisticCounting++. Essa função deve retornar a posição do bit ligado menos significativo de um inteiro. Assim, dado um inteiro x, tentaremos encontrar uma forma de encontrar o índice do bit ligado menos significativo de x. É interessante, portanto, considerar o oposto de x, ou seja, $-x$. Considerando a representação binária de x, $-x$ pode ser calculado da seguinte maneira: $\overline{x} + 1$, em que $\overline{x}$ é o complementar de x, cuja representação binária inverte todos os bits de x. Tendo $-x$ em mãos, a posição do único bit ligado de x & $-x$ coincide com a posição do bit ligado menos significativo de x. Para recuperarmos esse bit de x & $-x$, usamos a função `bit_length` do Python que retorna a menor quantidade de bits para se representar um inteiro. Esta função é indexada a partir do 1 e, por isso, subtraímos 1 do retorno dela.

Vamos ver alguns exemplos para deixar essa ideia mais clara. Tome $\rho(10) = \rho(0101_2) = 1$. Para chegar nesse valor de 1, primeiro encontramos $-10 = \overline{10}+1 = 1010_2+1000_2 = 0110_2$. Em seguida, calculamos $10 \,\&\, -10 = 0101_2 \,\&\, 0110_2 = 0100_2$. E a posição do único bit ligado deste número binário é 1. Outro exemplo é $\rho(9) = \rho(1001_2) = 0$. Assim, $-9 = \overline{9} + 1 = 0110_2 + 1000_2 = 1110_2$, e $9 \,\&\, -9 = 1001_2 \,\&\, 1110_2 = 1000_2$. E o único bit ligado deste número está na posição zero, que coincide com o valor de $\rho(9)$.

Outro detalhe de implementação é que não estamos utilizando a função de hash h durante a inserção dos elementos na estrutura, uma vez que, os elementos inseridos já são inteiros, e estamos supondo que eles são gerados uniformemente. Assim, não há a necessidade de mapeá-los para inteiros entre 0 e $2^L - 1$, inclusive. Ou seja, implicitamente, temos que h é a função identidade.

Tendo esclarecido os detalhes de implementação do Programa 3.1, podemos partir para a apresentação dos experimentos. O valor de L utilizado nesses experimentos foi de 64, uma vez que, a maior parte das linguagens de programação suporta atualmente inteiros de 8 bytes. Além disso, gerar número aleatórios de 64 bits garante uma maior variabilidade dos dados dos experimentos.

O primeiro experimento tem como objetivo verificar a evolução da estimativa devolvida pela estrutura conforme mais elementos são inseridos nela. Dessa forma, realizamos uma simulação em que geramos um milhão de inteiros aleatórios que foram sendo inseridos em uma estrutura `ProbabilisticCounting`. Essa simulação foi repetida duas vezes, sendo que na primeira, os elementos foram inseridos em um estrutura com $m = 64$ e na segunda vez, os mesmos itens foram adicionados à estrutura com $m = 1024$. Os resultados dessas simulações estão presentes na Figura 9. Analisando primeiramente, as estimativas da estrutura com $m = 64$, podemos notar que as estimativas, que são representadas pela linha laranja, ficaram levemente distantes dos valores reais, que são representados pela linha azul. Outro fato notável é que ao aumentarmos o valor de m para 1024, essa distância diminuiu, o que quer dizer que a precisão das estimativas aumentou.

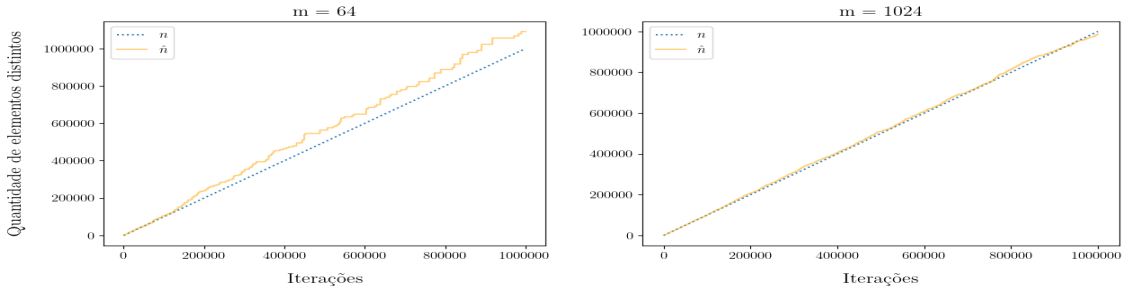


Figura 9: *Primeiro experimento do algoritmo ProbabilisticCounting++. Foram inseridos em estruturas* `ProbabilisticCounting` *com* $m = 64$ *e* $m = 1024$*, um milhão de inteiros de 64 bits gerados uniformemente.*

Na Seção 3.5, vimos que a Contagem Probabilística estima pequenas cardinalidades com um grande erro. Esse fato pode ser confirmado na Figura 10, que destaca as estimativas iniciais das estruturas com $m = 64$ e $m = 1024$. É possível notar que a linha das estimativas já começa distante da linha dos valores reais e que essa distância cresce quando o valor de m aumenta.

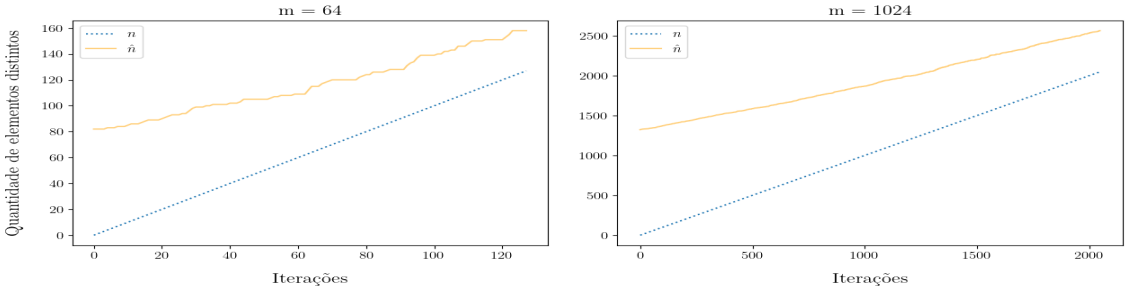


Figura 10: *Primeiro experimento do algoritmo ProbabilisticCounting++. Foram destacadas as primeiras 128 iterações da simulação com a estrutura* `ProbabilisticCounting` *com* $m = 64$ *e as 2048 estimativas iniciais da simulação com a estrutura com* $m = 1024$*. Podemos perceber que as estimativas iniciais apresentam um grande erro, que é a distância da linha laranja para a linha azul.*

Em vista desse erro no início do algoritmo, vamos desconsiderar as primeiras iterações do experimento anterior e tentar observar se `ProbabilisticCounting` começa a produzir melhores estimativas conforme mais itens são inseridos nessa estrutura. A Figura 11 exibe a evolução das estimativas devolvidas desconsiderando as primeiras. O desvio padrão da Contagem Probabilística é de $0{,}78 / \sqrt{m}$, e portanto, para $m = 64$, esse desvio é de 9,8%, e para $m = 1024$, 2,4%. Sendo assim, se desconsiderarmos as primeiras estimativas, a Contagem Probabilística com $m = 64$ devolveu valores dentro de dois desvios padrões. Já na simulação com $m = 1024$, as estimativas ficaram dentro de um desvio, indicando um aumento de precisão.

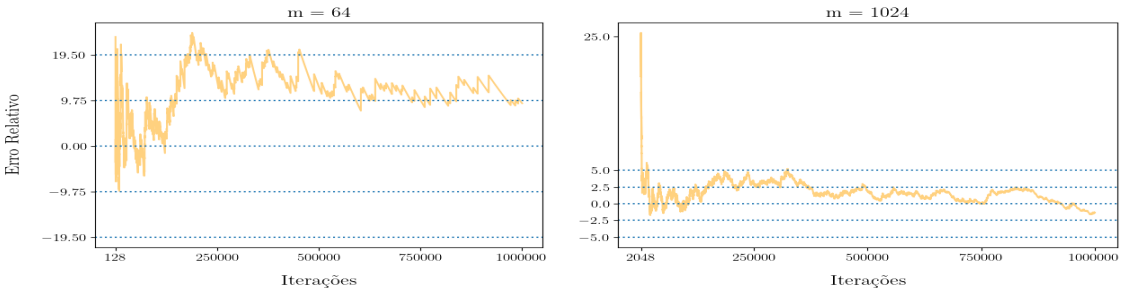


Figura 11: *Erro relativo do primeiro experimento do algoritmo ProbabilisticCounting++. Foram desconsideradas as primeiras 128 iterações da simulação com a estrutura* `ProbabilisticCounting` *com* $m = 64$ *e as 2048 estimativas iniciais da simulação com a estrutura com* $m = 1024$*. Podemos perceber que as estimativas da estrutura com* $m = 64$*, ficaram majoritariamente entre dois desvios padrões. Por outro lado, a maior parte dos valores devolvidos pela* Contagem Probabilística *com* $m = 1024$ *ficaram dentro de um desvio padrão.*

As duas simulações apresentadas anteriormente destacam a imprecisão da estrutura `ProbabilisticCounting` na hora de estimarmos pequenos valores e que aumentando o valor de m, essa imprecisão ficar ainda pior. Contudo, a partir de um certo ponto, esse problema desaparece e um parâmetro m maior implica em um algoritmo mais preciso. Dessa forma, o próximo experimento tem como objetivo verificar a variância da Contagem Probabilística e como m a influencia.

Assim como foi feito na Seção 2.4, repetimos as simulações do primeiro experimento

da Contagem Probabilística várias vezes, e coletamos a frequência das estimativas. A Figura 12 exibe como as estimativas devolvidas pela estrutura `ProbabilisticCounting` com $m = 64$ e $m = 1024$ ficaram distribuídas após dez mil simulações. Quase todas as estimativas ficaram entre dois desvios padrões nos dois casos, e os valores devolvidos pela estrutura com $m = 1024$ ficaram mais concentrados em torno da quantidade real de elementos distintos, que é de um milhão. Assim, quando aumentamos o valor de m, estamos diminuindo a variância do algoritmo.

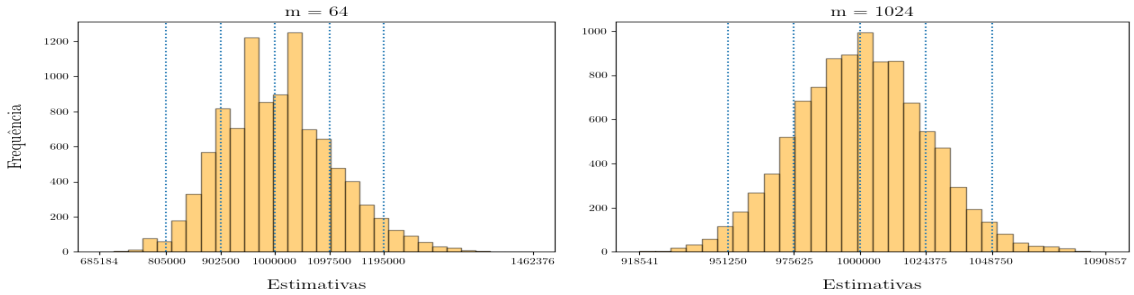


Figura 12: *Segundo experimento do algoritmo ProbabilisticCounting++. Foram realizadas dez mil simulações, e em cada simulação, foram inseridos em estruturas* `ProbabilisticCounting` *com* $m = 64$ *e* $m = 1024$*, um milhão de inteiros de 64 bits gerados uniformemente. Os resultados foram utilizados para a construção dos histogramas de frequência acima.*

A partir dos experimentos apresentados, foi possível termos noção da precisão do algoritmo da Contagem Probabilística e como manipular o parâmetro m para que tenhamos resultados melhores. Conseguimos, também, observar por meio das simulações a **não-linearidade inicial** desse algoritmo. Contudo, mesmo com esse obstáculo, essa estrutura de dados foi de grande importância para que pesquisas de bancos de dados pudessem ser otimizadas, e várias ideias dela serviram de base para que outras soluções com menor consumo de memória surgissem. E nos próximos capítulos, continuaremos vendo outros modos de resolvermos o problema da contagem distinta aproximada.

Capítulo 4

Contagens por amostragens

Este capítulo aborda uma outra solução para o problema da contagem distinta aproximada. O Capítulo 3 apresentou o algoritmo da Contagem Probabilística, que estima o número de elementos distintos em um fluxo de dados com desvio padrão em torno de $0{,}78 / \sqrt{m}$.

O algoritmo das Amostragens Adaptativas tem um desvio padrão de aproximadamente $1{,}20 / \sqrt{m}$ (Flajolet, 1990). As razões para estudarmos esse algoritmo mesmo que ele apresente um erro maior ficarão claras nas próximas seções.

4.1 Algoritmo das Amostragens Adaptativas

O algoritmo das Amostragens Adaptativas também se baseia em padrões de bits dos hashes dos elementos examinados. Podemos dessa forma, considerar que os elementos que estamos contando são palavras binárias de tamanho infinito.

Enquanto que no algoritmo da Contagem Probabilística, estávamos interessados na **aparição** de um elemento com prefixo da forma 0^*1, no algoritmo das Amostragens Adaptativas, queremos saber a **quantidade** de elementos com um certo prefixo 0^*.

Nesse sentido, vamos manter um contador δ cujo valor inicialmente é zero, e uma lista que armazena elementos com prefixo 0^δ. Para cada elemento examinado, inserimos ele na lista se o prefixo dele for da forma 0^δ. Quando essa lista tiver mais que m elementos, incrementamos δ em 1 e passamos a manter somente elementos com prefixo $0^{\delta+1}$.

Agora, suponha que em um dado momento do algoritmo, a lista tenha tamanho l e estamos mantendo somente elementos com prefixos 0^δ. A probabilidade de um elemento ter um prefixo dessa forma é $1/2^\delta$, e consequentemente, esperamos que a cada 2^δ itens, pelo menos um item tenha um prefixo desse formato. Assim, se essa lista tem l elementos com prefixo 0^δ, então devemos ter examinado pelo menos $2^\delta l$ itens, e este valor é justamente a estimativa para a quantidade de valores distintos.

O algoritmo AdaptiveSampling recebe um fluxo $\mathbb{M}$ com n elementos distintos e um parâmetro m, que está relacionado com o consumo de memória do algoritmo e sua precisão.

O algoritmo usa uma função de hash h para associar de forma uniforme os elementos de $\mathbb{M}$ a inteiros de L bits, e quanto maior for este valor, menor será a quantidade de colisões. Por fim, o valor devolvido é um estimador $\hat{n}$ para n da forma $2^\delta l$, em que l e δ indicam que estão sendo armazenados l elementos cujos prefixos têm formato 0^δ.

```
AdaptiveSampling(𝕄, m)
 1  LIST ← ∅
 2  δ ← 0
 3  para cada x em 𝕄
 4       se 0^δ é prefixo de h(x) e h(x) ∉ LIST
 5            LIST ← LIST ∪ {h(x)}
 6       enquanto |LIST| > m
 7            δ ← δ + 1
 8            TEMPLIST ← ∅
 9            para cada y em LIST
10                 se 0^δ é prefixo de h(y)
11                      TEMPLIST ← TEMPLIST ∪ {h(y)}
12            LIST ← TEMPLIST
13  devolva 2^δ|LIST|
```

4.2 Vantagens das Amostragens Adaptativas

Vamos simular um exemplo pequeno para ver o funcionamento do algoritmo AdaptiveSampling. Assim, queremos estimar quantos elementos distintos o fluxo $\mathbb{M} = \{36, 108, 41, 82, 71, 61, 5, 54, 10\}$ possui. Vamos considerar que a função de hash h é a identidade, ou seja, $h(x) = x$ para todo x em $\mathbb{M}$, e que $m = 3$.

Inicialmente, a lista $LIST$ está vazia e $\delta = 0$. Na primeira iteração do algoritmo, temos que verificar se o prefixo da representação binária do elemento $36 = 001001_2$ é da forma 0^0. Note que qualquer elemento passará nessa validação quando δ for zero, uma vez que, uma palavra binária precisar ter o prefixo 0^0 significa que ela tem que começar com 0 zeros, ou seja, pode começar com 0 ou com 1. Dessa forma, podemos inserir o elemento 36 na lista, de modo que $LIST = \{36\}$. Na próxima iteração, temos que fazer a mesma verificação anterior, só que usando o item de valor 108. Novamente, como $\delta = 0$, podemos adicionar 108 à lista. Analogamente, podemos incluir os elementos 41 e 82 em $LIST$.

Contudo, na quarta iteração, entramos no `while` da linha 6, já que $|LIST| = 4 > m = 3$. Por conta disso, incrementamos o valor de δ e devemos manter em $LIST$ somente aqueles valores cujos prefixos de suas representações binárias são da forma 0^1. Vamos ver quais elementos estão em $LIST$:

$$LIST = \{36 = 001001_2, 108 = 0011011_2, 41 = 100101_2, 82 = 0100101_2\}\,.$$

Precisamos manter nessa lista, somente aqueles números com prefixos que começam com 0. Assim, temos que remover o valor 41 de $LIST$.

Continuando o exemplo, precisamos verificar se adicionaremos $71 = 1110001_2$ na lista. Agora, os elementos em $LIST$ precisam começar com zero, logo 71 não é inserido. A mesma

situação ocorre para os valores $61 = 101111_2$ e $5 = 101_2$. Em seguida, podemos inserir $54 = 011011_2$ em $LIST$ e como novamente, $|LIST| = 4 > 3$, teremos que incrementar δ e filtrar os itens da lista. A lista atual é

$$\{36 = 001001_2, 108 = 0011011_2, 82 = 0100101_2, 54 = 011011_2\}.$$

Manteremos somentes os valores cujos prefixos binários se iniciam com dois zeros e portanto,

$$LIST = \{36 = 001001_2, 108 = 0011011_2\}.$$

Por fim, o valor $10 = 0101_2$ não é inserido na lista. E a estimativa para a quantidade de elementos distintos em $\mathbb{M}$ é $2^\delta \times |LIST| = 2^2 \times 2 = 8$. Essa estimativa está cometendo um erro de um item, já que $\mathbb{M}$ tem 9 itens distintos.

O fato curioso desse exemplo acontece nas primeiras iterações, quando δ tem valor zero. Vimos que qualquer elemento de $\mathbb{M}$ nessas primeiras iterações é inserido em $LIST$. Nessas situações, a estimativa do algoritmo é $2^\delta \times |LIST| = 2^0 \times |LIST| = |LIST|$. Ou seja, para os primeiros m itens distintos de $\mathbb{M}$, a estimativa tem precisão de 100%. Isto é o principal diferencial do algoritmo das Amostragens Adaptativas para o algoritmo da Contagem Probabilística, que como foi visto na Seção 3.5, apresenta um erro muito grande quando queremos estimar cardinalidades baixas.

Outro ponto importante de se destacar é que o estimador devolvido por AdaptiveSampling é **não-viesado** (Flajolet, 1990), ou seja, o valor esperado desse estimador coincide com a quantidade de elementos distintos que se quer estimar. Essa característica está relacionada com o fato do algoritmo das Amostragens Adaptativas poder estimar valores pequenos sem problemas.

4.3 Implementando Amostragens Adaptativas

Nesta seção apresentaremos a implementação do algoritmo das Amostragens Adaptativas e experimentos.

```python
class AdaptiveSampling:
    def __init__(self, m=64, L=64):
        self.delta: int = 0
        self.m: int = m
        self.L: int = L
        self.LIST: Set[int] = set()

    def zeros_no_prefixo(self, x: int):
        return (x & -x).bit_length() - 1

    def adiciona(self, x: int):
        if x not in self.LIST and self.zeros_no_prefixo(x) >= self.delta:
            self.LIST.add(x)

        while len(self.LIST) > self.m:
            self.delta = self.delta + 1
            temp: Set[int] = set()

            for y in self.LIST:
                if self.zeros_no_prefixo(y) >= self.delta:
                    temp.add(y)

            self.LIST = temp

    def conta(self):
        return (1 << self.delta) * len(self.LIST)
```

Programa 4.1: *Implementação do algoritmo AdaptiveSampling*

Vamos comentar alguns aspectos do código do Programa 4.1 baseado no algoritmo AdaptiveSampling. O primeiro detalhe de implementação desse programa é que a função de hash h é implicitamente a função identidade, uma vez que, estamos supondo que os elementos inseridos são inteiros gerados uniformemente. O próximo detalhe é a estrutura de dados utilizada para representar a lista *LIST*. Foi escolhido um Set para representá-la, que é uma estrutura padrão do Python que permite inserir elementos e verificar se um elemento está presente nela em tempo constante. Dessa forma, a complexidade do método `adiciona` é $O(1)$ amortizado.

Por fim, falta esclarecer como foi codificada a verificação se um inteiro tem prefixo da forma 0^δ, ou em outras palavras, se a representação binária de um inteiro começa com pelo menos δ zeros. Lembremos dessa forma, da função ρ definida na Seção 3.1. Essa função recebe um inteiro x e devolve a posição indexada a partir do zero do bit ligado menos significativo de x. Assim, $\rho(12) = \rho(0011_2) = 2$ e $\rho(24) = \rho(00011_2) = 3$. Note que a saída da função ρ corresponde também a quantidade de bits zeros no prefixo da representação binária de um inteiro. Podemos, portanto, aproveitar o método `p` da classe `ProbabilisticCounting` para implementar o método `zeros_no_prefixo` e agora, para

verificar se um inteiro x tem prefixo da forma $0^{\texttt{delta}}$, basta testar se `zeros_no_prefixo(x)` é maior ou igual a `delta`.

A partir deste ponto, apresentaremos experimentos do algoritmo das Amostragens Adaptativas, que serão os mesmos realizados na Seção 3.6, só que utilizando a estrutura apresentada nesse capítulo. Assim, o primeiro experimento que veremos é observar a evolução das estimativas devolvidas pela estrutura `AdaptiveSampling` conforme mais itens são inseridos nela. Portanto, nas simulações realizadas, inserirmos um milhão de inteiros de 64 bits distintos gerados uniformemente em estruturas com $m = 64$ e $m = 1024$. E o mesmo gerador de números pseudo-aleatórios dos experimentos do algoritmo da Contagem Probabilística foi utilizado, ou seja, os mesmos elementos foram aproveitados.

Os resultados do primeiro experimento podem ser observados na Figura 13. É possível perceber que a estrutura com o parâmetro m maior teve no geral, uma precisão melhor que a estrutura com m menor ao longo da simulação.

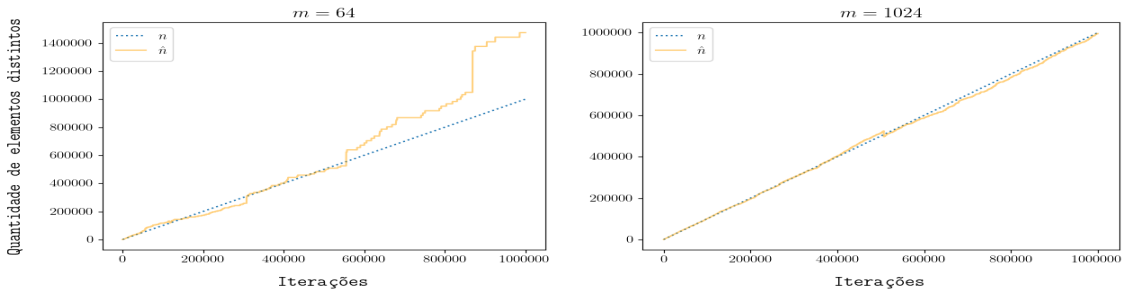


Figura 13: *Primeiro experimento do algoritmo AdaptiveSampling. Foram inseridos em estruturas* `AdaptiveSampling` *com* $m = 64$ *e* $m = 1024$*, um milhão de inteiros de 64 bits gerados uniformemente.*

Os gráficos com a evolução do erro relativo do experimento anterior estão na Figura 14. O desvio padrão do algoritmo das Amostragens Adaptativas é $1{,}20 / \sqrt{m}$. Portanto, para a estrutura com $m = 64$, o desvio padrão é de 15%. As estimativas devolvidas por essa estrutura ficaram entre dois desvios padrões na maior parte da simulação, apresentando um erro maior quando esta se aproximava do fim. Já a estrutura com $m = 1024$ tem desvio padrão de 3,75%, e a simulação dela teve valores que ficaram dentro de um desvio na maior parte das iterações.

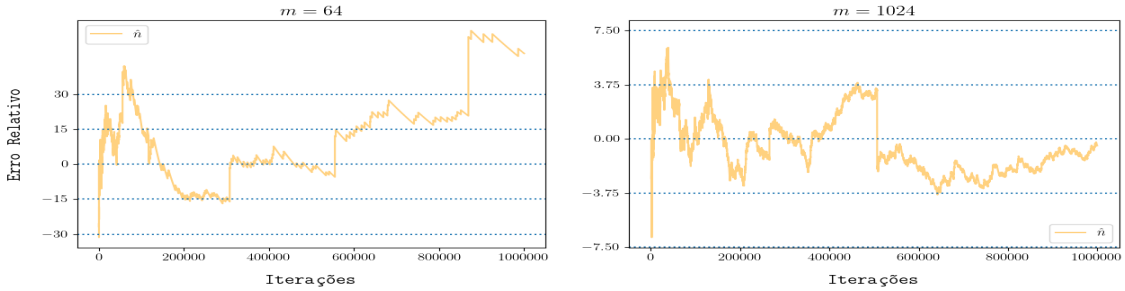


Figura 14: *Erro relativo do primeiro experimento do algoritmo AdaptiveSampling. Para* $m = 64$*, o erro ficou dentro de dois desvios padrões. Para* $m = 1024$*, a faixa de erro foi de um desvio padrão.*

Antes de passarmos para o segundo experimento, é interessante verificarmos se o que foi discutido na Seção 4.2 pode ser observado nas simulações anteriores. Queremos, assim, averiguar se o algoritmo das Amostragens Adaptativas funciona de fato para baixas cardinalidades. A Figura 15 tem gráficos que destacam o erro relativo nas primeiras 256 iterações no caso da estrutura com $m = 64$ e nas primeiras 4096 iterações no caso da estrutura com $m = 1024$. Nos dois casos, podemos notar que o erro relativo nas primeiras m iterações é de fato zero e que o erro após essas iterações iniciais se manteve controlado, dentro da faixa de dois desvios padrões.

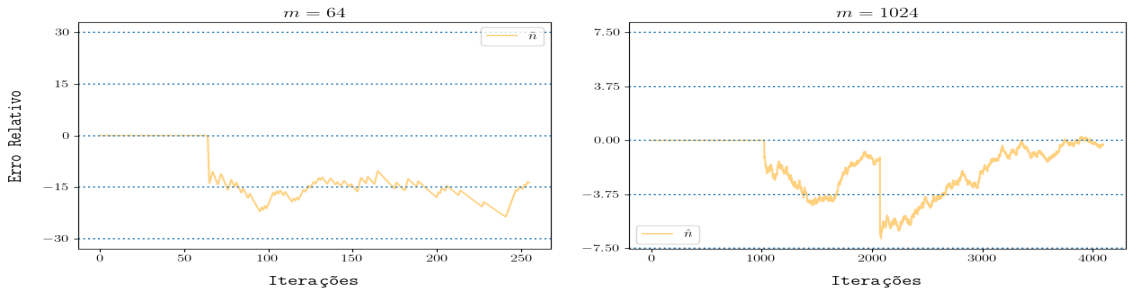


Figura 15: *Erro relativo nas primeiras iterações do experimento do algoritmo AdaptiveSampling. Nas m inserções iniciais,o erro relativo do algoritmo é zero.*

Para o segundo experimento, repetimos as simulações anteriores dez mil vezes. No final do processo, construímos histogramas a partir das frequências das estimativas devolvidas. A Figura 16 tem as distribuições das estimativas da estrutura `AdaptiveSampling` com $m = 64$ e $m = 1024$. Nos dois casos, quase todas as estimativas ficaram dentro de dois desvios padrões, e houve uma grande concentração de estimativas entre um desvio padrão. E podemos notar que com um valor de m maior, a variância do algoritmo das Amostragens Adaptativas diminuiu.

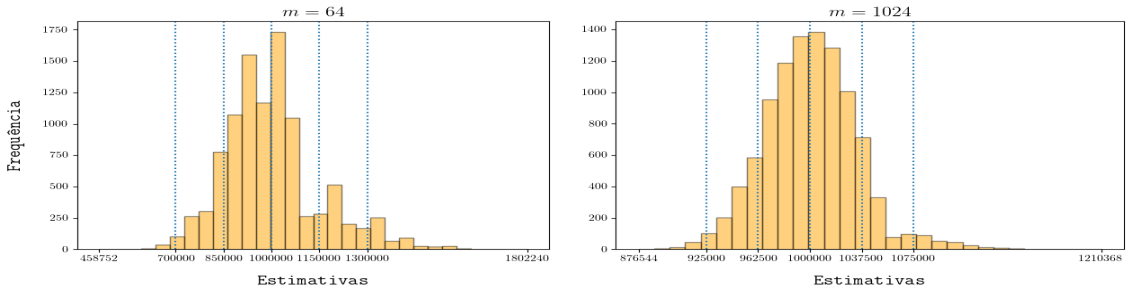


Figura 16: *Segundo experimento do algoritmo AdaptiveSampling. Foram realizadas dez mil simulações, e em cada simulação, foram inseridos em estruturas `AdaptiveSampling` com $m = 64$ e $m = 1024$, um milhão de inteiros de 64 bits gerados uniformemente. Os resultados foram utilizados para a construção dos histogramas de frequência acima.*

Com os experimentos apresentados nessa seção, foi possível termos noção da precisão do algoritmo das Amostragens Adaptativas. Este algoritmo tem pontos interessantes, como se basear em padrões de bits para construir a estimativa e funcionar bem para fluxos com poucos elementos. Porém, a principal desvantagem dessa solução é o consumo de

memória. Se considerarmos uma aplicação que precise lidar com hashes de 32 bits, então o consumo de memória da estrutura `AdaptiveSampling` seria de pelos menos $32 \times m$ bits no pior caso, quando a lista *LIST* passa a ter mais que m elementos. Esse custo é o mesmo que o do algoritmo Contagem Probabilística quando $L = 32$.

Para que esse gasto de memória fosse reduzido sem que se perdesse muita precisão, demorou mais de 20 anos desde a criação da Contagem Probabilística. Veremos no próximo capítulo como podemos reduzir ainda mais esse consumo.

Capítulo 5

LogLog's

Em 1978, Morris resolveu o problema de estimar quantos elementos passaram por um fluxo de dados sem armazenar esse valor em um contador. E as ideias principais da solução dele era guardar uma aproximação do logaritmo dessa quantidade e incrementá-la por meio de um método probabilístico que lembra lançamentos de moedas. Dessa forma, se temos um fluxo com n elementos, e X é a aproximação do algoritmo de Morris para $\lg n$, então X deve ser incrementado com probabilidade $1 \,/\, 2^X$ para cada novo elemento, e isto é equivalente a lançarmos uma moeda X vezes e aumentar X somente se todas as jogadas forem cara.

Essa solução inspirou o desenvolvimento do algoritmo da Contagem Probabilística, que resolvia o problema de estimar a quantidade de elementos distintos que passaram por um fluxo $\mathbb{M}$. Nesse algoritmo, estamos interessados em usar padrões de bits de números inteiros aleatórios para gerar essa estimativa. Portanto, os dados de $\mathbb{M}$ são mapeados para inteiros por meio de uma função de hash, e assim, podemos considerar que esses elementos são na verdade palavras binárias de tamanho infinito. Essa simplificação nos permite agrupar os itens de $\mathbb{M}$ por prefixos da forma 0^*1. A razão de fazermos isto é que a probabilidade de a representação binária de um inteiro ter um prefixo da forma $0^{X-1}1$ é $1 \,/\, 2^X$, a mesma que aparece no algoritmo de Morris, e portanto, essa ideia pode ser usada para estimarmos o logaritmo da quantidade de elementos distintos em $\mathbb{M}$ da seguinte forma: guardaremos o menor R tal que ainda não tenha aparecido algum inteiro com prefixo 0^R1.

Dessa maneira, para um fluxo $\mathbb{M}$ com n elementos distintos, R seria a estimativa para $\lg n$. No entanto, por meio do cálculo do valor esperado desse estimador, é possível concluir que R, na verdade, não estima $\lg n$, mas $\lg \phi n$ (Flajolet e Martin, 1985). Em outras palavras, o algoritmo da Contagem Probabilística produz uma estimativa com um desvio ϕ mensurável e podemos, em vista disso, corrigi-lo.

Contudo, mesmo corrigindo esse erro do estimador, este ainda possui uma grande variância, ou seja, a estimativa devolvida pode estar muito próxima ou muito longe do valor real. Para contornar essa situação, podemos repetir várias vezes o algoritmo da Contagem Probabilística e encontrar a média dos estimadores. E uma forma interessante de se fazer isso em uma única iteração, removendo assim, a necessidade de executar-

mos o algoritmo muitas vezes, é dividir de modo uniforme os dados do fluxo em m lotes. Então, um inteiro x faria parte do lote $x \bmod m$, e olharíamos para o prefixo de $\lfloor x \,/\, m \rfloor$. Desse modo, passaríamos a guardar um estimador R para cada lote, e a estimativa para n usaria a média desses valores. Essa técnica é conhecida como **média estocástica**.

Por fim, o algoritmo da Contagem Probabilística tem consumo proporcional a $O(mL)$ bits, em que m é o número de lotes e L é a quantidade de bits necessária para armazenar algum inteiro do fluxo. Esse consumo é decorrente do fato de cada lote precisar guardar a informação se um prefixo da forma 0^*1 já apareceu, e podemos fazer isto com L bits por lote. Nesse sentido, o próximo algoritmo, chamado LogLog, terá como base muitas ideias da Contagem Probabilística, e apresentará uma significativa redução do consumo de espaço.

5.1 Algoritmo LogLog

Dado um fluxo $\mathbb{M}$ com n elementos distintos, o algoritmo LogLog devolverá um estimador $\hat{n}$ para n, que terá um desvio padrão de $1{,}30/\sqrt{m}$ (Durand e Flajolet, 2003). Assim como foi feito no algoritmo da Contagem Probabilística, precisamos ter uma função de hash h que mapeie uniformemente os dados de um fluxo para inteiros. Tendo esta função em mãos, podemos supor que estamos trabalhando com um fluxo de inteiros aleatórios. E vamos manter o maior M tal que exista algum elemento em $\mathbb{M}$ cuja representação binária tenha prefixo da forma $0^{M-1}1$.

Para $M \geq 1$, a probabilidade de um inteiro ter um prefixo da forma $0^{M-1}1$ é $1 \,/\, 2^M$, ou seja, esperamos que um em cada 2^M elementos tenha esse prefixo. Por outro lado, podemos inverter essa ideia, e pensar que se temos um inteiro com prefixo da forma $0^{M-1}1$, então pelo menos 2^M elementos distintos devem ter aparecido. Desse modo, o valor de M será uma aproximação para $\lg n$, e podemos interpretá-lo como sendo a posição indexada do 1 do bit ligado menos significativo de um inteiro. Uma ideia similar foi vista na Seção 3.1, em que definimos a função ρ que retorna a posição indexada do 0 do bit ligado menos significativo de um número. Em vista disso, vamos definir a função ρ_1 que retorna o valor anterior só que indexado do 1, e utilizá-la para o cálculo de M.

Para reduzirmos a variância da estimativa, dividimos o fluxo $\mathbb{M}$ em $m = 2^k$ lotes de acordo com os bits dos elementos. Dessa forma, um inteiro $x = \langle b_1 b_2 \ldots b_k \ldots \rangle$ fará parte do lote $y = \langle b_1 b_2 \ldots b_k \rangle$, ou seja, os k primeiros bits de um elemento indicam em qual lote ele pertence. Em seguida, encontramos M_y tal que $0^{M_y}1$ é prefixo de $\langle b_{k+1} b_{k+2} \ldots \rangle$, e mantemos o maior valor de M_y.

Desse modo, o valor de M_y de cada lote y será uma estimativa para $\lg \frac{n}{m}$. Para diminuirmos a variância dessa aproximação, calculamos a média **aritmética** $\overline{M} = \frac{\sum_y M_y}{m}$. Logo, $\overline{M}$ aproxima $\lg \frac{n}{m}$, e consequentemente, $2^{\overline{M}}$ aproxima $\frac{n}{m}$. E, portanto, a estimativa para n será $m \times 2^{\overline{M}}$.

No entanto, essa estimativa apresenta um desvio e para corrigí-lo, multiplicamos ela por uma constante α_m cujo valor é proporcional ao número de lotes utilizados no algoritmo. Na prática, podemos usar como fator de correção $\alpha_\infty \approx 0{,}39701$ assim que m for maior que 64.

Em vista disso, a saída do algoritmo LogLog será um estimador $\hat{n}$ da forma $\alpha_m \times m \times 2^{\overline{M}}$. E o pseudocódigo a seguir condensa as ideias apresentadas.

LogLog$\left(\mathbb{M}, m = 2^k\right)$
1 **para** i de 0 até m
2 $\quad M[i] \leftarrow 0$
3 **para** cada x em $\mathbb{M}$
4 $\quad b_1 b_2 ... \leftarrow h(x)$
5 $\quad lote \leftarrow \langle b_1 ... b_k \rangle$
6 $\quad M[lote] \leftarrow \max(M[lote], \rho_1(b_{k+1} b_{k+2} ...))$
7 **devolva** $\alpha_m \times m \times 2^{\frac{1}{m}\sum_i M[i]}$

Vamos simular um exemplo pequeno para entendermos com mais detalhes o algoritmo LogLog. O objetivo, portanto, é estimar a quantidade de elementos distintos em $\mathbb{M} = \{50, 85, 45, 29, 89, 82, 87, 10, 92\}$. Suponhamos que $k = 2, m = 2^k = 2^2 = 4$, que a função de hash h seja a função identidade e que $\alpha_m = \alpha_4 = 1$, ou seja, estamos desconsiderando o desvio da estimativa devolvida pelo algoritmo.

Inicialmente, os vetores M estão zerados, ou seja, $M[i] = 0$ para $0 \leq i < 4$. Na primeira iteração do algoritmo, temos que verificar em qual lote o elemento $50 = 010011_2$ se encontra. Como $k = 2$, os primeiros 2 bits definem o lote e assim, 50 faz parte do lote $01_2 = 2$. Ao final da primeira iteração, temos que $M[2] = 3$, já que $\rho_1(0011_2) = 3$ e dessa forma, 0011_2 tem prefixo da forma $0^2 1$

Na segunda iteração, o número do lote de $85 = 1010101_2$ é 1, e $\rho_1(10101_2) = 1$. Desse modo, $M[1] = 1$. Os próximos elementos a serem considerados, $45 = 101101_2$ e $29 = 10111_2$, fazem parte do lote de número 1, e têm prefixo da forma $0^0 1$. Dessa maneira, $M[1]$ continua sendo um após esses itens serem processados. Precisamos, assim, analisar $89 = 1001101_2$, que pertence ao lote 1. Como $\rho_1(01101_2) = 2$, $M[1]$ passa a ter valor 2.

O próximo elemento de $\mathbb{M}$ é $82 = 0100101_2$ cujo lote é 2. Temos que $\rho_1(00101) = 3$, mas o valor de M do lote 2 já é 3, sendo assim, $M[2]$ continua com o mesmo valor. Continuando o exemplo, devemos processar $87 = 1110101_2$. Este item está no lote 3, e $\rho_1(10101_2) = 1$. Logo, $M[3]$ passa a ser igual a 1. No caso seguinte, $10 = 0101_2$ faz parte do lote 2 e $\rho_1(01_2) = 2$. Contudo, $M[2]$ é maior que 2 e portanto, não é atualizado. Por fim, $92 = 0011101_2$ vai para o lote 0, $\rho_1(11101_2) = 1$ e $M[0] = 1$.

Os valores de M ao processarmos todos os elementos de $\mathbb{M}$ são: $M[0] = 1$, $M[1] = 2$, $M[2] = 3$ e $M[3] = 1$. Logo, o valor médio $\overline{M}$ de M é $(1 + 2 + 3 + 1)/4 = 7/4 = 1{,}75$. E a estimativa para a quantidade de itens distintos em $\mathbb{M}$ é $\alpha_m \times m \times 2^{\overline{M}} = 1 \times 4 \times 2^{1{,}75} = 13{,}45... \approx 13$.

5.2 Consumo de espaço do LogLog

No algoritmo da Contagem Probabilística, vimos a ideia de dividirmos os elementos de um fluxo de dados em m lotes. E como cada lote armazena um vetor bmap com L bits, o consumo de espaço dessa solução é de $O(mL)$ bits. Assim, para $m = 1024$ e $L = 32$,

a Contagem Probabilística gasta 4 KB de memória para estimar o número de itens distintos em um fluxo.

Por outro lado, o algoritmo LogLog também divide os dados de um fluxo $\mathbb{M}$ com n elementos distintos em m lotes. Só que ao invés de um lote guardar a informação se um prefixo já apareceu ou não, ele armazena uma aproximação para $\lg n$. Logo, se para representarmos um inteiro x, precisamos de pelo menos $\lg x$ bits, para guardar $\lg n$, são necessários $\lg \lg n$ bits. Este fato deu origem ao nome do algoritmo.

Considerando o exemplo anterior em que $m = 1024 = 2^{10}$ e que os dados do fluxo são inteiros de 32 bits, os 10 primeiros bits definem o lote de um elemento e os 22 restantes serão utilizados para encontrar M. Dessa forma, temos que $M \leq 22$ e precisamos de somente 5 bits para guardar esse valor. Portanto, o custo de espaço do algoritmo LogLog nesse caso é de 0,625 KB, que é cerca de seis vezes menor que na Contagem Probabilística.

Essa diferença é maior ainda se trabalharmos com inteiros de 64 bits e valores de m na ordem de $65536 = 2^{16}$ para termos estimativas mais precisas. No algoritmo LogLog, M pode ser armazenado com 6 bits, pois $M \leq 48$, e desse modo, o custo de memória é de 50 KB, dez vezes menos que na Contagem Probabilística, que gasta aproximadamente 524 KB.

5.3 Implementando LogLog

```python
class LogLog:
    def __init__(self, m=64):
        self.m = m
        self.B = math.floor(math.log2(m))
        self.M = [0] * m
        self.Z = 0
        self.alpha = 0.39701

    def p(self, x: int):
        return (x & -x).bit_length()

    @property
    def prefixo(self):
        return (1 << self.B) - 1

    def adiciona(self, x: int):
        lote = x & self.prefixo
        w = x >> self.B

        if self.p(w) > self.M[lote]:
            self.Z -= self.M[lote]
            self.M[lote] = self.p(w)
            self.Z += self.M[lote]

    def conta(self):
        Z_media = self.Z / self.m
        return math.floor(self.alpha * self.m * math.pow(2.0, Z_media))
```

Programa 5.1: *Implementação do algoritmo LogLog*

A classe `LogLog` do Programa 5.1 foi baseada no algoritmo LogLog. Vamos fazer alguns comentários sobre o método `adiciona`. O primeiro deles é relacionado à identificação do lote de um elemento x. Nessa implementação, a variável `B` faz o papel da variável k do algoritmo LogLog. Vimos na Seção 5.1, que os k primeiros bits definem o lote. Logo, os `B` bits iniciais contém a informação do lote de x. Sabendo disto, precisamos de um modo de recortar esses bits.

Um jeito de fazermos isto é montarmos um número tal que os primeiros `B` bits dele sejam 1 e os restantes, 0. Vamos chamá-lo de y. Nesse sentido, x & y resultará em um inteiro z cuja representação binária tem as seguintes propriedades: os `B` primeiros bits correspondem aos `B` bits iniciais de x, e os bits restantes são zero. Portanto, z é justamente o lote de x. Falta, assim, saber como calcular y. Um número com as mesmas características de y é $2^{\texttt{B}} - 1$, e podemos calculá-lo usando o operador <<, que é equivalente à potenciação na base 2 quando aplicado ao número 1.

Um exemplo que pode deixar a ideia anterior clara é o seguinte: suponha que $\texttt{B} = 2$. Logo, $2^{\texttt{B}} - 1 = 2^2 - 1 = 3 = 11_2$. Note que todos os bits de 3 estão ligados até a posição 2. Tomemos alguns valores para x, como $7 = 111_2$ e $13 = 1011_2$. O lote do elemento 7 é $7 \,\&\, 3 = 111_2 \,\&\, 110_2 = 110_2 = 3$, e de 13, $13 \,\&\, 3 = 1011_2 \,\&\, 1100_2 = 1000_2 = 1$. Podemos ver que os lotes calculados batem de fato com os valores esperados.

Uma vez calculado o lote de um elemento x, precisamos remover os `B` bits iniciais dele para que possamos prosseguir com o algoritmo. Um modo de fazermos isto é utilizando o operador >> que remove os bits menos significativos de um inteiro. Assim, seja $w = x >> \texttt{B}$. A representação binária de w será a mesma que a de x, só que sem os primeiros `B` bits. Logo, tomando novamente $B = 2$ e $x = 13 = 1011_2$, w seria igual a $1011_2 >> 2 = 11_2$.

Por fim, para que possamos ter um método `conta` com complexidade constante, precisamos ter a soma Z dos valores de M já pré-computada para que a média deles seja calculada rapidamente. Dessa forma, a variável Z é mantida atualizada a cada novo elemento inserido no método `adiciona`.

5.4 Algoritmo HyperLogLog

Apesar do algoritmo LogLog visto anteriormente ter um consumo de memória pelo menos 6 vezes menor que a Contagem Probabilística, o desvio padrão dele, que é de $1{,}30 / \sqrt{m}$, é maior. Isto pode ser um problema se tivermos que trabalhar com quantidades muito grandes, da ordem de bilhões. No entanto, com uma simples modificação no algoritmo LogLog, obtemos um novo algoritmo com o mesmo consumo de memória, mas desvio padrão de $1{,}04 / \sqrt{m}$.

O aprimoramento do algoritmo LogLog é conhecido como HyperLogLog (Flajolet, Fusy *et al.*, 2007). E a modificação que deve ser feita em LogLog é substituir a média **aritmética** pela média **harmônica**. A intuição por trás desta substituição é que a média harmônica é afetada menos por *outliers*, ou seja, valores muito fora do esperado não distorcem tanto a estimativa. A consequência disso é que a variância do estimador é reduzida.

Tanto no algoritmo LogLog quanto no HyperLogLog, o valor de M de cada lote é uma aproximação para $\lg n / m$, e conseguimos obter uma estimativa para n / m se elevarmos 2 a esse valor. Dessa forma, no algoritmo LogLog, a média aritmética das aproximações de $\lg n / m$ é calculada. Por outro lado, no algoritmo HyperLogLog, calculamos a média harmônica das aproximações de n / m. Podemos encontrar a média $\overline{M}$ da seguinte forma:

$$\overline{M} = \left(\frac{\sum \left(2^{M[i]}\right)^{-1}}{m} \right)^{-1} = \frac{m}{\sum 2^{-M[i]}}$$

O fato de trocarmos a média aritmética pela harmônica também interfere no fator de correção α_m. Na prática, podemos usar a aproximação $\alpha_\infty \approx 0{,}7213$ assim que m for maior que 128 (*HyperLogLog* 2022). Por fim, o algoritmo HyperLogLog traduz essas ideias para pseudocódigo, e é possível notar que ele é praticamente idêntico ao algoritmo LogLog. A única diferença entre essas soluções está na última linha.

HyperLogLog($\mathbb{M}, m = 2^k$)

1 **para** i de 0 até m
2 $\quad M[i] \leftarrow 0$
3 **para** cada x em $\mathbb{M}$
4 $\quad b_1 b_2 \ldots \leftarrow h(x)$
5 $\quad lote \leftarrow \langle b_1 \ldots b_k \rangle$
6 $\quad M[lote] \leftarrow \max(M[lote], \rho_1(b_{k+1} b_{k+2} \ldots))$
7 **devolva** $\alpha_m \times m^2 \,/\, \sum_i 2^{-M[i]}$

5.5 Implementando HyperLogLog

```python
class HyperLogLog:
    def __init__(self, m=64):
        self.m = m
        self.B = math.floor(math.log2(m))
        self.M = [0] * m
        self.Z = self.m
        self.alpha = 0.7213

    def p(self, x: int):
        return (x & -x).bit_length()

    @property
    def prefixo(self):
        return (1 << self.B) - 1

    def adiciona(self, x: int):
        lote = x & self.prefixo
        w = x >> self.B

        if self.p(w) > self.M[lote]:
            self.Z -= math.pow(2, -self.M[lote])
            self.M[lote] = self.p(w)
            self.Z += math.pow(2, -self.M[lote])

    def conta(self):
        return math.floor(self.alpha * self.m * self.m / self.Z)
```

Programa 5.2: *Implementação do algoritmo HyperLogLog*

O Programa 5.2 foi baseado no algoritmo HyperLogLog e é quase idêntico ao Programa 5.1, que foi inspirado no algoritmo LogLog. Na implementação do LogLog, a variável `Z` era a soma dos valores de `M`. Como no HyperLogLog, a média aritmética é substituída pela harmônica, a variável `Z` passa a guardar outro tipo de soma. Em vista disso, o formato do estimador também muda, e consequentemente, o método `conta` da classe `HyperLogLog` é ligeiramente diferente do mesmo método da classe `LogLog`.

Essa troca de médias também afeta o método `adiciona`. As implementações apresentadas retornam a estimativa da quantidade de elementos distintos em tempo constante, e assim, a cada novo item inserido no método `adiciona`, devemos atualizar a variável `Z` para evitar computá-la toda vez que o método `conta` for chamado. E é justamente essa atualização da variável `Z` que difere nos dois programas.

Capítulo 6

Contagem na vida real

Nos últimos capítulos, vimos algumas soluções do problema de estimar a quantidade de elementos distintos de um fluxo de dados. A solução mais usada atualmente é a estrutura de dados HyperLogLog.

O Redis, que é um banco de dados de chave-valor salvo em memória, oferece uma implementação dessa estrutura de dados. Nessa estrutura, precisamos definir um valor de m, que é um parâmetro relacionado à precisão das estimativas e ao consumo de memória. No Redis, esse valor é de $2^{14} = 16384$. Assim, o desvio padrão dessa implementação é de $1{,}04 / \sqrt{m} = 1{,}04 / \sqrt{16384} \approx 0{,}81\%$.

A função de hash utilizada na implementação do Redis produz saídas de 64 bits. Como $m = 14$, os valores que são considerados nos lotes do HyperLogLog tem 50 bits, e são necessários somente 6 bits para guardar a informação de cada lote. Em vista disso, são armazenados 16 mil lotes que consomem 6 bits cada, e o consumo da estrutura fica em torno de 12 KB.

As aplicações que utilizam alguma solução do problema da contagem distinta aproximada precisam geralmente, manter quantos elementos distintos apareceram em **vários** fluxos de dados. A consequência disso é que surge a necessidade de por exemplo, salvarmos uma estrutura HyperLogLog em um banco de dados para evitar termos todas as estruturas desse tipo em memória. Nesse sentido, o Programa 5.2 está incompleto, pois em um primeiro momento, não existe uma forma fácil de guardarmos a classe `HyperLogLog` em um banco de dados.

A implementação do Redis contorna esse problema separando o código dos métodos `adiciona` e `conta` do armazenamento dos dados dos lotes (*Redis new data structure: the HyperLogLog* 2014). Por conta disso, a estrutura HyperLogLog do Redis é somente uma sequência de bits que comprime a informação dos lotes. E com essa representação em bits, podemos guardar essa estrutura no banco de dados.

6.1 Quantas visitas um site teve?

Uma das métricas interessantes que aplicativos e sites podem coletar é a quantidade de acessos diários. A importância desse tipo de informação é que ela pode ajudar a responder perguntas como "Tenho muitos acessos nos fins de semana?" ou "Depois do lançamento de uma certa funcionalidade, o número de acessos aumentou?".

Contudo, pode existir situações em que é importante identificar o número de acessos distintos. Esse dado pode ser interessante para lojas virtuais ou redes sociais, uma vez que, essas aplicações dependem que várias pessoas diferentes acessem esse tipo de site ou aplicativo.

Para resolvermos esse problema, podemos aproveitar o endereço de IP dos usuários que acessarem os servidores de um site. Dessa forma, estaríamos trabalhando com um fluxo de endereços, e queremos descobrir quantos IP's distintos apareceram em um certo dia. Portanto, inserimos os IP's que forem acessando o servidor em uma estrutura HyperLogLog, e ao final do dia, salvamos a estimativa no banco de dados e zeramos o HyperLogLog. Assim, esse banco teria a informação dos acessos distintos diários. E essa ideia pode ser estendida para obtermos quantos acessos diferentes um site teve em uma certa hora do dia.

6.2 Quantas pessoas leram um artigo?

O problema anterior exigiu o uso de apenas uma estrutura HyperLogLog. Porém, existem situações nas quais mais estruturas desse tipo precisam ser utilizadas. Um desses casos é contar quantas visualizações um dado artigo teve.

Reddit é uma plataforma cuja principal funcionalidade é permitir a criação de canais de discussões sobre diversos assuntos. Uma métrica interessante, portanto, é quantos usuários distintos visualizaram uma discussão específica. Apesar de esse site não estar coletando atualmente essa métrica por conta de problemas de performace (*We disabled the view count feature :(* 2018), a solução apresentada pela equipe de engenharia dele exibe uma situação em que várias estruturas HyperLogLog são utilizadas (*View Counting at Reddit* 2017).

Os engenheiros do Reddit utilizaram a implementação do HyperLogLog disponível no Redis. Para entendermos os próximos passos, é necessário termos noção de como funciona esse banco de dados. O Redis é basicamente um conjunto de chaves e valores, e o programador tem a sua disposição alguns métodos para interagir com esse conjunto. Um desses métodos é o `SET`, que recebe os parâmetros `key` e `value` e adiciona a essa coleção esse par de chave e valor. Se já existir um par com a chave `key`, o valor antigo é substituído por `value`.

Outro comando útil é o `PFADD`, que recebe uma chave `key` e uma palavra. Este método insere elementos em uma estrutura HyperLogLog e retorna 1 se o item for de fato inserido ou 0, caso contrário. Nesse sentido, o Redis supõem que o valor da chave `key` é uma sequência de bits que representa um HyperLogLog. Caso a chave `key` não exista no momento da inserção, uma estrutura vazia é criada. E também existe o comando `PFCOUNT`

que recebe uma chave `key` e retorna a estimativa para a quantidade de dados diferentes inseridos na estrutura HyperLogLog, que está armazenada no valor desta chave.

Em vista disso, a solução do Reddit armazena várias estruturas HyperLogLog em uma instância do Redis, de modo que cada estrutura está associada a uma discussão. Contudo, o Redis é um banco de dados salvo em memória, e não é possível guardar todas as estruturas de uma vez. Para contornar isso, as estruturas são salvas em disco, mais especificamente em um banco de dados não-relacional chamado Cassandra. Dessa forma, o Redis só terá as estruturas cujas respectivas discussões foram acessadas recentemente.

De modo simplificado, para cada nova visualização que precisar ser contada, precisamos verificar se a estrutura associada à discussão visualizada está presente no Redis. Se estiver, basta chamarmos o método `PFADD` passando como argumentos o identificador da discussão como chave e o identificador do usuário, que pode ser o endereço IP, como valor. E caso o retorno de `PFADD` seja 1, indicando que o contador foi incrementado, devemos atualizar a nova contagem no Cassandra.

Se a estrutura HyperLogLog da discussão não existir no Redis, devemos procurá-la no Cassandra e criar uma nova caso ela não exista. Em seguida, devemos carregá-la no Redis, e o modo de se fazer isso é chamar o método `SET` passando como parâmetro o identificador da discussão como chave e o HyperLogLog como valor. Por fim, os mesmos passos da situação em que a discussão já está no Redis devem ser seguidos.

Por fim, devemos consultar o Cassandra para sabermos quantas visualizações distintas uma discussão específica teve.

Capítulo 7

Conclusão

Ao longo desse texto, vimos algumas soluções do problema da contagem distinta aproximada. Um fato comum entre elas é que todas dependem de um parâmetro m que afeta a acurácia e o consumo de memória dos algoritmos. Outro ponto é que todas utilizam alguma função de hash que mapeia os elementos de um fluxo de dados para inteiros. E a quantidade de bits destes interfere também no consumo de espaço. Em vista disso, supondo que essa quantidade é de 32 bits, a Tabela 7.1 compara diferentes estruturas de dados que resolvem esse problema de contagem.

Algoritmo	Desvio Padrão	Consumo de memória por lote
ProbabilisticCounting	$0{,}78 / \sqrt{m}$	32 bits
AdaptiveSampling	$1{,}20 / \sqrt{m}$	32 bits
LogLog	$1{,}30 / \sqrt{m}$	≤ 5 bits
HyperLogLog	$1{,}04 / \sqrt{m}$	≤ 5 bits

Tabela 7.1: *Soluções da contagem distinta aproximada. Estamos supondo as saídas das funções de hash são inteiros de 32 bits.*

A base desses algoritmos é tentar manter uma aproximação de $\lg n$, sendo que n é a quantidade de elementos distintos em um fluxo de dados. O custo de memória para mantermos uma aproximação desse tipo é da ordem de $O(\lg n)$ bits, que representa uma grande redução se compararmos com o gasto de espaço linear da solução que insere todos os elementos em uma tabela.

Encontrar um estimador para $\lg n$ é possível por conta de padrões nos bits de números aleatórios. Contudo, essa estimativa tem uma grande variância, e para contornar esse problema, dividimos uniformemente o fluxo de dados em m lotes.

Sendo assim, a Contagem Probabilística é um dos algoritmos com menor desvio padrão até os dias de hoje. No entanto, o algoritmo HyperLogLog se tornou a principal solução para o problema da contagem distinta aproximada, devido ao seu consumo de memória seis vezes menor que o da Contagem Probabilística.

Referências

[Andoni 2017] Alexandr Andoni. *Lecture 2 – Approximate Counting and Hashing*. 2017. url: https://www.mit.edu/~andoni/s17_advanced/algorithms/mainSpace/files/scribe2.pdf (acesso em 22/08/2021) (citado nas pgs. 7, 13).

[Bell *et al.* 1990] Timothy C. Bell, John G. Cleary e Ian H. Witten. *Text Compression*. USA: Prentice-Hall, Inc., 1990 (citado na pg. 5).

[Durand e Flajolet 2003] Marianne Durand e Philippe Flajolet. "Loglog counting of large cardinalities". Em: (2003). url: http://algo.inria.fr/flajolet/Publications/DuFl03-LNCS.pdf (citado nas pgs. 2, 46).

[Flajolet 1990] Philippe Flajolet. "On adaptive sampling". Em: *Computing* 43 (1990), pgs. 391–400. url: https://link.springer.com/article/10.1007%2FBF02241657 (acesso em 11/10/2021) (citado nas pgs. 2, 37, 39).

[Flajolet, Fusy *et al.* 2007] Philippe Flajolet, Éric Fusy, Olivier Gandouet e Frédéric Meunier. "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm". Em: (2007). url: http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf (citado nas pgs. 2, 50).

[Flajolet e Martin 1985] Philippe Flajolet e Nigel Martin. "Probabilistic counting algorithms for data base applications". Em: *Journal of Computer and System Sciences* 31 (1985), pgs. 182–209. url: https://dsf.berkeley.edu/cs286/papers/flajoletmartin-jcss1985.pdf (acesso em 30/08/2021) (citado nas pgs. 2, 23, 24, 26, 30, 31, 45).

[*HyperLogLog* 2022] *HyperLogLog*. url: https://en.wikipedia.org/wiki/HyperLogLog (acesso em 13/02/2022) (citado na pg. 50).

[Lumbroso 2018] Jérémie O. Lumbroso. "The story of hyperloglog: how flajolet processed streams with coin flips". Em: *arXiv: Data Structures and Algorithms* (2018) (citado na pg. 5).

[McMahon *et al.* 1978] L. E. McMahon, L. L. Cherry e R. Morris. "Unix time-sharing system: statistical text processing". Em: *The Bell System Technical Journal* 57.6 (1978), pgs. 2137–2154 (citado na pg. 5).

[Morris 1978] Robert Morris. “Counting large numbers of events in small registers”. Em: *Communications of the ACM* 21 (1978), pgs. 840–842 (citado nas pgs. 2, 5).

[Morris e Cherry 1975] Robert Morris e Lorinda L. Cherry. “Computer detection of typographical errors”. Em: *IEEE Transactions on Professional Communication* PC-18 (1975), pgs. 54–56 (citado na pg. 5).

[*Redis new data structure: the HyperLogLog* 2014] *Redis new data structure: the HyperLogLog*. 2014. url: http://antirez.com/news/75 (acesso em 13/02/2022) (citado na pg. 53).

[*View Counting at Reddit* 2017] *View Counting at Reddit*. 2017. url: https://redditblog.com/2017/05/24/view-counting-at-reddit (acesso em 30/08/2021) (citado na pg. 54).

[*We disabled the view count feature :(* 2018] *We disabled the view count feature :(* 2018. url: https://www.reddit.com/r/changelog/comments/a74eor/we_disabled_the_view_count_feature/ (acesso em 14/02/2022) (citado na pg. 54).

[Wikipedia 2021] Wikipedia. *Approximate counting algorithm*. url: https://en.wikipedia.org/wiki/Approximate_counting_algorithm (acesso em 22/08/2021) (citado na pg. 6).