Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

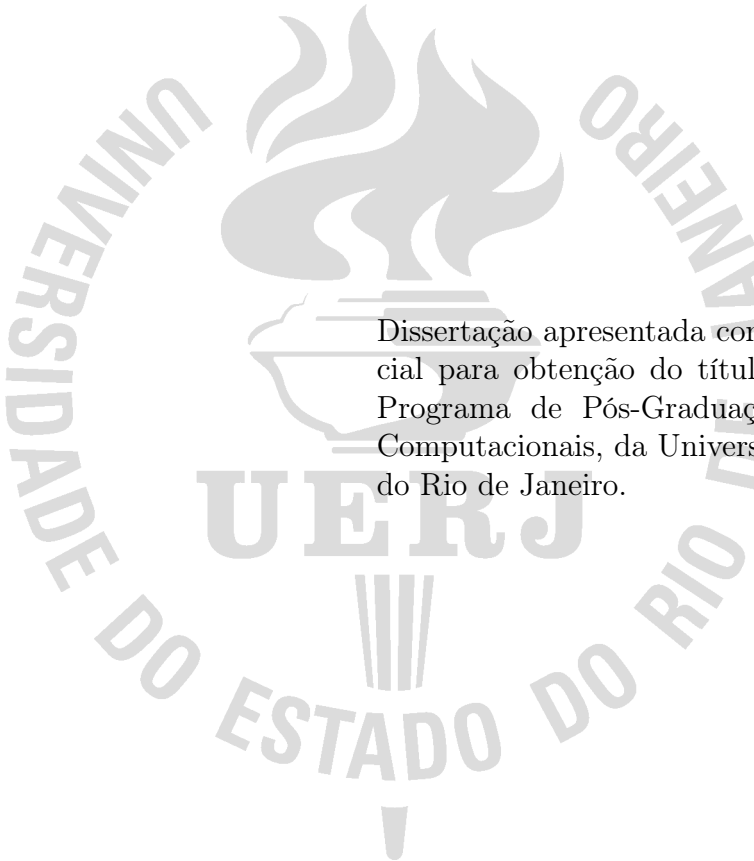
Juan Pedro Alves Lopes

Estruturas de dados probabilísticas aplicadas à representação implícita de grafos

Rio de Janeiro
2017

Juan Pedro Alves Lopes

Estruturas de dados probabilísticas aplicadas à representação implícita de grafos



Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Orientador: Prof. Dr. Paulo Eustáquio Duarte Pinto
Coorientador: Prof. Dr. Fabiano de Souza Oliveira

Rio de Janeiro
2017

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC-A

L864 Lopes, Juan Pedro Alves.
Estruturas de dados probabilísticas aplicadas à representação implícita de grafos/ Juan Pedro Alves Lopes. – 2017.
102 f. : il.

Orientador: Paulo Eustáquio Duarte Pinto
Coorientador: Fabiano de Souza Oliveira
Dissertação (Mestrado em Ciências Computacionais)- Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística.

1. Grafos – Teoria - Teses. I. Pinto, Paulo Eustáquio Duarte. II. Oliveira, Fabiano de Souza. III. Universidade do Estado do Rio de Janeiro. Instituto de Biologia Roberto Alcantara Gomes. IV. Título.

CDU 519.17

_____ _____

Assinatura Data

Juan Pedro Alves Lopes

Estruturas de dados probabilísticas aplicadas à representação implícita de grafos

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Aprovada em 10 de Março de 2017
Banca Examinadora:

Prof. Dr. Paulo Eustáquio Duarte Pinto (Orientador)
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Fabiano de Souza Oliveira (Coorientador)
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Jayme Luiz Szwarcfiter
Universidade Federal do Rio de Janeiro

Prof. Dr. Valmir Carneiro Barbosa
Universidade Federal do Rio de Janeiro

Prof. Dr. Igor Machado Coelho
Instituto de Matemática e Estatística - UERJ

Rio de Janeiro
2017

DEDICATÓRIA

À minha esposa, Jacqueline,
e ao meu filho, Miguel.

RESUMO

LOPES, Juan Pedro Alves. *Estruturas de Dados Probabilísticas Aplicadas à Representação Implícita de Grafos*. 2017. 102 f. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro.

Esta dissertação apresenta duas principais contribuições. É feito um resumo da literatura sobre quatro estruturas de dados probabilísticas importantes: *Bloom filters*, *Count-Min sketch*, *MinHash* e *HyperLogLog*. São discutidas suas definições, variantes, limites de erro e aplicações práticas. Novos dados experimentais sobre essas estruturas foram produzidos para este trabalho. Além disso, é discutida aqui a aplicação de estruturas de dados probabilísticas para o problema de representação de grafos. Duas novas representações probabilísticas são introduzidas aqui: uma que usa filtros de Bloom e pode representar grafos gerais com a mesma complexidade da matriz de adjacência (sendo até melhor para grafos esparsos), e outra que usa *MinHash* e pode representar árvores com complexidade menor que a representação determinística ótima.

Palavras-chave: Estruturas de dados probabilísticas, grafos, representação de grafos.

ABSTRACT

LOPES, Juan Pedro Alves. *Probabilistic data structures applied to implicit graph representation.* 2017. 102 f. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro.

This thesis comprises two major contributions. It summarizes the literature about four important probabilistic data structures: *Bloom filters*, *Count-Min sketch*, *MinHash*, and *HyperLogLog*, discussing their definition, variants, error bounds, and practical applications, while providing new experimental data about them. Also, it discusses the application of probabilistic data structures to the graph representation problem. Two new probabilistic representations are introduced here: one that uses Bloom filters and can represent general graphs with the same complexity as the adjacency matrix (but outperforms it for sparse graphs), the other that uses MinHash and can represent trees with lower complexity than the optimal deterministic implicit representation.

Keywords: Probabilistic data structures, graphs, graph representation.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

INTRODUÇÃO

O rápido aumento da capacidade computacional nas últimas décadas impulsionou a criação de novos sistemas que consomem imensos volumes de dados em um fluxo contínuo. Exemplos destes sistemas incluem aplicações financeiras, sistemas de telemetria de equipamentos e servidores, redes de sensores em plantas de produção, navios ou plataformas de petróleo, análise de transações em aplicações web, dentre muitas outras.

Nestes sistemas, muitas vezes não é viável carregar todos os dados em um SGBD relacional para posteriormente efetuar as consultas necessárias. Normalmente a latência obtida nesta abordagem não é suficiente para o tipo de respostas que se deseja obter. As consultas precisam ser executadas continuamente sobre os dados que chegam. Historicamente, os SGBDs não permitem a execução de este tipo de consultas [TGNO92]. Consultas contínuas diferem de consultas tradicionais por manterem um certo estado em memória e emitirem potencialmente mais de um resultado, sempre que alguma condição para tal for atingida. Alguns exemplos de tipos de consultas incluem:

- **consultas periódicas:** os resultados da consulta devem ser atualizados após a passagem de um certo período de tempo;
- **consultas de transformação:** os resultados da consulta devem ser atualizados sempre que um novo registro é adicionado ao sistema;
- **consultas de eventos complexos:** os resultados da consulta devem ser atualizados sempre que um padrão complexo é detectado, por exemplo: o recebimento de dois registros seguidos com menos de 10 segundos de diferença entre eles.

Neste contexto, surge a necessidade de um novo modelo de processamento de dados, onde as consultas operam apenas sobre novos registros recebidos pelo sistema. Esses registros geralmente são imutáveis, porém fazem parte de fluxos de dados potencialmente infinitos. Outros desafios envolvem a imprevisibilidade sobre a ordem ou taxa de chegada desses registros, bem como a inviabilidade de analisar a qualquer momento os registros recebidos anteriormente. Algoritmos que funcionam em um modelo de fluxo de dados geralmente precisam ser capazes de computar resultados com apenas uma passagem sobre os dados [BBD+02].

Como os fluxos de dados podem ser virtualmente infinitos, a quantidade de memória necessária para consultas arbitrárias sobre eles também tende a crescer indefinidamente. Embora algoritmos que executam eficientemente fora da memória principal venham sendo estudados ao longo das últimas décadas, estes algoritmos não se mostram apropriados para uso em aplicações em tempo real, por geralmente não suportarem consultas contínuas e

por terem desempenho aquém do requerido para aplicações com alto volume de eventos. Por este motivo, focaremos em algoritmos que usam apenas uma quantidade finita da memória principal para efetuar suas operações [AMS96].

Alguns problemas podem ter algoritmos trivialmente adaptados para funcionar neste modelo de processamento, como o cálculo da média. Outros, entretanto, podem não ser tão adequados. Um exemplo é o problema de calcular a mediana (ou qualquer quantil) em um fluxo de números inteiros, pois o mesmo requer mais de uma passagem pelos dados ou que eles sejam armazenados para computar o resultado. Em [MP80] mostra-se que calcular quantis em uma série com N valores em p passagens pelos dados requer $\Omega(N^{\frac{1}{p}})$ espaço.

Neste trabalho, abordaremos estruturas de dados conhecidas como estruturas de sinopse (*synopsis*) ou esboço (*sketch*) [GM99]. Iremos nos referir a elas apenas como estruturas de dados probabilísticas. A principal característica destas é utilizar uma quantidade limitada de recursos. Em especial:

- podem residir na memória principal, evitando operações de E/S;
- permitem atualização incremental;
- consomem menos recursos que os dados originais consumiriam, se fossem armazenados;
- podem ser facilmente transmitidas ou armazenadas;
- servem como substitutos de baixo custo dos dados originais, viabilizando alguns tipos de consultas.

Frequentemente, estruturas de dados probabilísticas descartam algumas das informações que seriam necessárias para representar fielmente os dados originais e permitir uma resposta exata às consultas. Estas estruturas de dados aproximam a resposta com um fator de erro relativo à quantidade de recursos destinados ao processamento.

Em especial, este trabalho foca nas estruturas que se aproveitam da codificação em *hash* dos elementos de um conjunto ou multiconjunto para responder consultas específicas.

O trabalho está organizado da seguinte maneira: no Capítulo 1, introduziremos os conceitos básicos de probabilidade e funções *hash* necessários para compreender as estruturas que serão apresentadas nas seções seguintes.

Ao longo do Capítulo 2 abordamos quatro estruturas probabilísticas. A característica comum entre elas é o uso de funções *hash* para representar probabilisticamente certos conjuntos. Cada estrutura permite um certo conjunto de operações.

- Na Seção 2.1, abordamos o filtro de Bloom [Blo70], uma estrutura de dados que responde consultas aproximadas de pertinência de elementos em um conjunto utilizando menos memória do que o necessário para armazenar o conjunto inteiro. É

característica dos filtros de Bloom que as respostas negativas sejam exatas, enquanto as respostas positivas sejam aproximadas.

- Na Seção 2.2, consideramos a estrutura probabilística *Count-Min* [CM05], que é análoga aos filtros de Bloom, permitindo estimar não só a pertinência, mas também a frequência de um elemento em um multiconjunto. Analogamente, qualquer frequência estimada pela estrutura é garantidamente maior ou igual à frequência real do elemento no multiconjunto. É também possível empregá-la para estimar quantis em um fluxo de dados.

- Na Seção 2.3 introduzimos o *MinHash* [Bro97], estrutura que permite estimar a semelhança entre dois conjuntos sem precisar compará-los elemento a elemento.

- Na Seção 2.4, abordamos a estrutura *HyperLoglog* [FFGM08], que responde a cardinalidade aproximada de elementos distintos de um fluxo de dados utilizando memória adicional constante.

No Capítulo 3 introduzimos a teoria sobre representações implícitas de grafos e apresentamos aplicações e ideias para representações implícitas probabilísticas baseadas em algumas estruturas apresentadas neste trabalho.

Por fim, a Conclusão resume as contribuições deste trabalho, sugerindo novos trabalhos futuros envolvendo representações de grafos utilizando estruturas de dados probabilísticas.

1 CONCEITOS BÁSICOS

A fim de facilitar o entendimento das ideias exploradas nos próximos capítulos, introduzimos aqui os conceitos básicos utilizados ao longo deste trabalho.

Em especial, abordaremos o básico de probabilidade, explicando desde o conceito de espaço amostral até variáveis aleatórias e estimadores; conceitos essenciais para explicar e demonstrar as propriedades de estruturas de dados probabilísticas.

Além disso, faremos uma revisão do estado da arte sobre funções *hash*, criptográficas e não-criptográficas, falando também sobre *hashing* universal e terminando com alguns exemplos de funções *hash* importantes.

1.1 Introdução à probabilidade

Estruturas de dados probabilísticas claramente dependem muito da teoria e notações da cálculo de probabilidades. Neste trabalho, a fim de facilitar o entendimento, introduzimos o assunto utilizando notação definida em [FFLS07].

1.1.1 Espaço probabilístico

Define-se o *espaço probabilístico* como formado por dois componentes: um *espaço amostral* $\Omega = \{r_1, r_2, \ldots\}$, que representa os possíveis resultados de um experimento aleatório, e uma *função de probabilidade* Pr, que define a probabilidade com que qualquer *evento* $A \subseteq \Omega$ ocorre em experimentos aleatórios. Esta função precisa respeitar as seguintes propriedades:

1. Para todo evento $A \subseteq \Omega$, $0 \leq \Pr[A] \leq 1$.
2. $\Pr[\Omega] = 1$.
3. Para eventos $A_1, A_2, ..., A_n$ disjuntos, $\Pr[\bigcup_i A_i] = \sum_i \Pr[A_i]$.

No caso de eventos $A_1, A_2, ..., A_n$ arbitrários (isto é, não necessariamente disjuntos), vale o *princípio da inclusão-exclusão*:

$$\begin{aligned}\Pr\left[\bigcup_i A_i\right] = &\sum_i \Pr[A_i] \\ &- \sum_{i<j} \Pr[A_i \cap A_j] \\ &+ \sum_{i<j<k} \Pr[A_i \cap A_j \cap A_k] \\ &- \cdots \\ &+ (-1)^{l+1} \sum_{i_1<i_2<\cdots<i_l} \Pr\left[\bigcap_{r=1}^{l} A_{i_r}\right] \\ &+ \cdots\end{aligned}$$

e em especial

$$\Pr[A_1 \cup A_2] = \Pr[A_1] + \Pr[A_2] - \Pr[A_1 \cap A_2]$$

Como exemplo destes conceitos, podemos analisar as probabilidades envolvidas no lançamento de um dado de seis lados. Definimos $\Omega = \{1, 2, 3, 4, 5, 6\}$, representando todos os possíveis resultados deste experimento.

Sabemos que a probabilidade de qualquer resultado $E_i = \{r_i\}, r_i \in \Omega$ é igual, isto é $\Pr[E_i] = {}^1/_6$. Podemos calcular que a probabilidade de um lançamento resultar em 2 ou 5 é dada pela probabilidade da união entre os dois eventos que, para conjuntos disjuntos, é igual a: $\Pr[\{2, 5\}] = \Pr[\{2\}] + \Pr[\{5\}] = {}^2/_6 = {}^1/_3$.

Entretanto, para calcular a probabilidade do resultado ser um número divisível por dois (evento Div2) ou por três (evento Div3) podemos utilizar o princípio da inclusão-exclusão. Assim

$$\Pr[\textsc{Div2} \cup \textsc{Div3}] = \Pr[\textsc{Div2}] + \Pr[\textsc{Div3}] - \Pr[\textsc{Div2} \cap \textsc{Div3}]$$

ou seja,

$$\begin{aligned}\Pr[\{2, 4, 6\} \cup \{3, 6\}] &= \Pr[\{2, 4, 6\}] + \Pr[\{3, 6\}] - \Pr[\{6\}] \\ &= {}^3/_6 + {}^2/_6 - {}^1/_6 \\ &= {}^2/_3\end{aligned}$$

1.1.2 Variáveis aleatórias

Uma *variável aleatória* X é uma função que mapeia o espaço amostral em um outro conjunto. Neste trabalho apenas lidaremos com variáveis aleatórias reais, isto é, na forma

$X : \Omega \to \mathbb{R}$.

A função *massa de probabilidade* $p_X(x) : \mathbb{R} \to [0, 1]$ de uma variável aleatória real X é definida como $p_X(x) = \Pr[X = x]$, que é outra forma de escrever $\Pr[\{r \in \Omega \mid X(r) = x\}]$, ou seja, a probabilidade do resultado r de um experimento aleatório seja tal que $X(r) = x$.

O *valor esperado* (ou *esperança*) de uma variável aleatória consiste na média de todos os possíveis valores que ela pode assumir ponderada pela probabilidade que cada valor tem de ser assumido. Isto é,

$$\mathrm{E}[X] = \sum_{x \in \mathbb{R}} x p_X(x).$$

Por exemplo, considere o resultado do lançamento de dois dados de seis lados. Podemos definir a variável X como a soma dos valores obtidos nos dados. Assim, temos por exemplo que $p_X(3) = {}^2/_{36}$, pois apenas dois resultados possíveis neste experimento resultam na soma 3 ($1 + 2$ e $2 + 1$). Por outro lado, $p_X(7) = {}^6/_{36}$, pois seis resultados possíveis no espaço amostral somam 7 ($1+6$, $2+5$, etc.). Assim, o valor esperado para a variável X é

$$\begin{aligned}
\mathrm{E}[X] &= 2 \cdot p_X(2) + 3 \cdot p_X(3) + ... + 12 \cdot p_X(12) \\
&= 2 \cdot {}^1/_{36} + 3 \cdot {}^2/_{36} + ... + 12 \cdot {}^1/_{36} \\
&= 7
\end{aligned}$$

A *linearidade da esperança* é a propriedade que diz que, para variáveis aleatórias $X_1, X_2, \cdots, X_n$ e uma função linear h, vale

$$\mathrm{E}[h(X_1, X_2, \cdots, X_n)] = h(\mathrm{E}[X_1], \mathrm{E}[X_2], \cdots, \mathrm{E}[X_n]).$$

Ainda utilizando o exemplo do lançamento de dois dados, uma outra forma de calcular o valor esperado seria fazê-lo para apenas um dado e dobrar o resultado. Considere as variáveis aleatórias Y_1 e Y_2 como os valores resultantes do lançamento de cada um dos dados, tal que $X = Y_1 + Y_2$. Assim, pela linearidade da esperança, $\mathrm{E}[X] = \mathrm{E}[Y_1 + Y_2] = \mathrm{E}[Y_1] + \mathrm{E}[Y_2]$.

O valor esperado para Y_i é mais fácil de calcular, sendo apenas a média simples entre todos os resultados possíveis no lançamento de um dado:

$$\mathrm{E}[Y_i] = \frac{1 + 2 + 3 + 4 + 5 + 6}{6} = \frac{21}{6} = \frac{7}{2}$$

logo

$$\mathrm{E}[X] = \mathrm{E}[Y_1] + \mathrm{E}[Y_2] = \frac{7}{2} + \frac{7}{2} = 7$$

Uma propriedade importante de variáveis aleatórias é sua variância, que pode ser

definda da seguinte forma:

$$\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - (\mathrm{E}[X])^2$$

Dadas variáveis aleatórias independentes $X_1, X_2, \cdots, X_n$, todas com a mesma variância σ^2, a *fórmula de Bienaymé* estabelece que a variância da média $\bar{X}$ dessas variáveis é dada por

$$\mathrm{Var}[\bar{X}] = \mathrm{Var}\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n^2}\sum_{i=1}^{n}\mathrm{Var}[X_i] = \frac{\sigma^2}{n}$$

Isto significa, na prática, que a média entre n variáveis independentes diminui a variância por um fator de n. Este resultado é útil para estruturas de dados probabilísticas, pois permite diminuir o erro compondo estimadores independentes.

Algumas variáveis aleatórias são muito comumente vistas na teoria e possuem muitas propriedades já estudadas. Para este trabalho iremos focar na variável de *Bernoulli* e na variável *binomial*, sendo a primeira apenas um caso especial da segunda.

A variável de Bernoulli pode assumir apenas dois valores: 0 ou 1. Ela representa o resultado de um experimento onde o valor 0 representa um "fracasso" e o valor 1 um "sucesso". Seja p a probabilidade de sucesso, temos que

$$p_X(x) = \begin{cases} 1-p & \text{se } x = 0 \\ p & \text{se } x = 1 \end{cases}$$

É fácil demonstrar que para uma variável de Bernoulli X, $\mathrm{E}[X] = p$ e $\mathrm{Var}[X] = p(1-p)$.

A variável binomial representa o total de sucessos em uma série de n experimentos e pode ser vista como o somatório de n variáveis de Bernoulli com uma mesma probabilidade p. Denota-se $B(n,p)$ a variável binomial que representa o total de sucessos de n experimentos com probabilidade p. A massa de probabilidade desta variável é

$$p_{B(n,p)}(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x} & \text{para } 0 \leq x \leq n \\ 0 & \text{para demais valores de } x \end{cases}$$

Perceba que a variável de Bernoulli é apenas um caso especial de uma variável binomial, $B(1,p)$.

O valor esperado da variável binomial pode ser inferido pela linearidade da esperança de cada variável de Bernoulli que a compõem, resultando em $E[B(n,p)] = np$.

A partir da definição $\mathrm{E}[X^2] = n(n-1)p^2 + np$, podemos calcular a variância da variável binomial: $\mathrm{Var}[X] = np(1-p)$.

1.1.3 Desigualdades probabilísticas

Embora o valor esperado de uma variável aleatória nos forneça a informação sobre como uma variável se comporta na média de vários experimentos, ele não nos diz muito sobre a distribuição dos valores que a variável pode assumir. Ao utilizar estruturas de dados probabilísticas, por exemplo, calculamos o valor de variáveis aleatórias cujo valor esperado é igual ao que se deseja estimar. Entretanto, para ter alguma utilidade prática, é preciso poder prever a probabilidade de erro. A seguir apresentaremos algumas desigualdades que permitem definir um limite superior na probabilidade da variável aleatória assumir certos intervalos de valores.

A *desigualdade de Markov* estabelece que para variáveis X que somente assumem valores não-negativos,

$$\Pr[X \geq a] \leq \frac{\mathrm{E}[X]}{a}, \text{ para todo } a > 0.$$

Se a variância da variável for conhecida, um limite mais forte pode ser estabelecido, através da *desigualdade de Chebyshev*, dada por

$$\Pr[|X - \mathrm{E}[X]| \geq a] \leq \frac{\mathrm{Var}[X]}{a^2}, \text{ para todo } a > 0.$$

É possível reescrever esta desigualdade definindo o desvio padrão $\sigma = \sqrt{\mathrm{Var}[X]}$. Assumindo $a = k\sigma$,

$$\Pr[|X - \mathrm{E}[X]| \geq k\sigma] \leq \frac{1}{k^2}, \text{ para todo } k > 0.$$

Este limite mostra, de forma intuitiva, como as probabilidades de cada um dos valores de X se comportam conforme eles se afastam de $\mathrm{E}[X]$.

Sem mais informações sobre a variável não é possível definir limites gerais mais fortes para sua distribuição. Entretanto, para variáveis compostas pela soma de variáveis independentes é possível utilizar a *desigualdade de Chernoff* para encontrar limites específicos da distribuição. Existem diversas variantes desta desigualdade, cada uma específica para um tipo de variável aleatória. Em especial, abordaremos aqui o uso desta desigualdade para variáveis de Bernoulli.

Considere uma variável $X = \sum_{i=1}^{n} X_i$, onde todo X_i é uma variável de Bernoulli independente com probabilidade p_i. Seja também $\mu = \mathrm{E}[X] = \sum_{i=1}^{n} p_i$. Então, dois limites se aplicam:

- **Limite superior:** $\Pr[X \geq (1+\delta)\mu] \leq e^{-\frac{\delta^2}{2+\delta}\mu}$ para todo $\delta > 0$;
- **Limite inferior:** $\Pr[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2}$ para todo $0 < \delta < 1$;

Uma versão simplificada que combina combina os dois limites é definida por

$$\Pr[|X - \mu| \geq \delta\mu] \geq 2e^{-\mu\delta^2/3} \text{ para todo } 0 < \delta < 1.$$

1.1.4 Estimadores

Um *estimador* é uma variável aleatória que permite estimar um certo parâmetro θ sobre outra variável X a partir de observações sobre a mesma. São funções denotadas $\hat{\theta} : E \to \Theta$, onde E é a imagem da variável X e Θ é chamado de espaço paramétrico. Estimadores, por serem variáveis aleatórias, possuem média, variância, desvio padrão etc.

O *viés* $\mathrm{B}(\hat{\theta})$ de um estimador é a diferença entre seu valor esperado e o parâmetro que ele almeja estimar, ou seja, $\mathrm{B}(\hat{\theta}) = \mathrm{E}[\hat{\theta}] - \theta$. Um estimador é considerado *não-enviesado* se $\mathrm{B}(\hat{\theta}) = 0$. O *erro padrão* de um estimador é definido como $\sigma_{\hat{\theta}} = \sqrt{\mathrm{Var}[\hat{\theta}]}$. O *erro médio quadrático* (EMQ) de um estimador combina variância e viés em um único conceito. Em vez de comparar a diferença entre cada observação e o valor esperado do estimador, compara-se com o valor real do parâmetro. Isto é $\mathrm{EMQ}[\hat{\theta}] = \mathrm{E}[(\hat{\theta} - \theta)^2]$. Em um estimador não-enviesado, o EMQ é igual à variância.

1.2 Funções *hash*

Todas as estruturas descritas neste trabalho são baseadas em funções *hash* e são, de certa forma, dependentes das propriedades das funções utilizadas. Por isso, apresentamos aqui um breve resumo das características principais esperadas de funções aplicáveis a estruturas de dados probabilísticas.

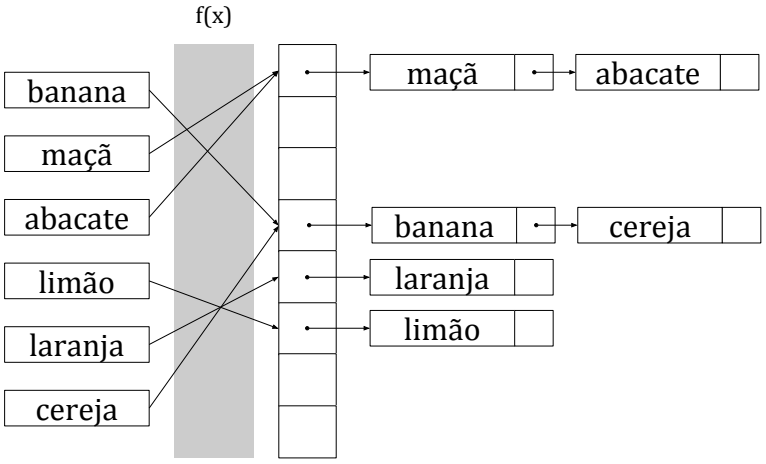
O objetivo principal das funções *hash* neste contexto é a capacidade de simular certa aleatoriedade para um conjunto de dados que pode ser altamente estruturado. É impossível definir uma função que cria dados aleatórios a partir de dados não aleatórios [Knu98], mas na prática é possível criar uma imitação boa o suficiente.

1.2.1 Tabelas *hash*

Originalmente, a aplicação principal de funções *hash* é a possibilidade de mapear chaves contidas em conjuntos muito grandes a entradas numa tabela com tamanho potencialmente bem menor do que os conjuntos que representam, uma tabela *hash*. Por exemplo, é possível mapear *strings*, um conjunto infinito, para índices na tabela, de forma que a complexidade esperada para verificar sua existência seja independente do tamanho da tabela, e sim proporcional ao tamanho da *string*, desde que o número de entradas na tabela seja um fator do número de elementos inseridos. A Figura 1 mostra um exemplo deste tipo de tabela.

O exemplo dado assume algumas propriedades da função *hash* escolhida, em especial:

Figura 1: Exemplo de tabela *hash*



- **A distribuição da imagem é aparentemente uniforme:** isto é, a probabilidade aparente de qualquer um dos valores do contradomínio ser escolhido para representar a entrada é igual. Esta é uma propriedade impossível de alcançar na teoria e sua aplicação prática pode ser amplamente dependente do domínio da função. Por exemplo: uma função *hash* que funciona bem para palavras em português pode não funcionar tão bem para palavras em inglês.

- **A função tem baixa complexidade (linear, de preferência):** Funções com complexidades maiores podem tornar a inserção e busca na tabela inviáveis.

- **A função é determinística:** esta é uma propriedade redundante, se assumirmos a definição usual de *função*, entretanto dado o contexto é importante explicitá-la, visto que uma função *hash* que resulta em valores diferentes (se aplicada várias vezes à mesma entrada) não tem valor para busca em tabelas. Dito isto, os conceitos de funções pseudo-aleatórias e funções *hash* possuem grande interseção teórica.

Há outros detalhes teóricos e de implementação de tabelas *hash* que não serão tratados aqui (como a resolução de colisões por exemplo), por não dizerem tanto a respeito de funções *hash* em si, que são o foco desta seção.

1.2.2 *Hashes* criptográficos

Não há um conjunto ideal de propriedades de funções *hash* que sirva a todos os propósitos possíveis. As propriedades ideais para uso em uma tabela *hash* não são as mesmas para o uso em aplicações de segurança, por exemplo. Nestas aplicações, geralmente espera-se que satisfaça-se algumas propriedades adicionais, de forma a garantir que usuários maliciosos não possam explorar as propriedades da construção da função para obter dados não autorizados do sistema.

Espera-se que uma função *hash* criptográfica h possua as seguintes propriedades adicionais [KL14]:

- **Resistência a pré-imagem:** Dado um hash $y = h(x)$, é computacionalmente inviável encontrar x' tal que $h(x') = y$. Isto é, um adversário não conseguiria inverter a função h, de forma a encontrar mensagens que resultem no *hash* dado.

- **Resistência a segunda pré-imagem:** Dada uma mensagem x, é computacionalmente inviável encontrar $x' \neq x$ tal que $h(x') = h(x)$. Isto é, um adversário não conseguiria forjar uma mensagem que resulta no mesmo hash que a mensagem dada.

- **Resistência a colisão:** É computacionalmente inviável encontrar quaisquer pares $(x, x'), x' \neq x$ tal que $h(x) = h(x')$. Isto é, um adversário não conseguiria encontrar duas mensagens distintas que resultem no mesmo hash.

As propriedades acima estão dispostas em ordem de robustez a ataques de adversários. Na prática é muito difícil satisfazer todas elas, entretanto, quanto mais as propriedades são satisfeitas, mais segura se torna a função *hash*.

Um corolário destas propriedades é conhecido como *Efeito Avalanche*, que garante que pequenas mudanças nas mensagens produzam grandes efeitos no resultado da função *hash*. Isto é especialmente importante para evitar que um adversário seja capaz de determinar a semelhança entre duas mensagens apenas observando seus *hashes*.

Funções *hash* criptográficas possuem muitos usos na prática. Entre eles, podemos citar:

- **Representação segura de senhas:** Consiste em usar funções *hash* para codificar e verificar igualdade entre senhas sem precisar armazenar as senhas em si, valendo-se do fato de que se $x = y \implies h(x) = h(y)$. Este método é utilizado na maior parte dos serviços na *web*, para evitar que um vazamento acidental do banco de dados acabe tornando públicas as senhas dos usuários. Geralmente esses *hashes* são melhorados com uso de uma técnica chamada *salting*, onde outros dados são combinados à chave a ser embaralhada pela função para evitar outros tipos de vazamento de informação. Por exemplo, é possível incluir o ID do usuário na chave para evitar que dois usuários com a mesma senha possuam o mesmo *hash*.

- **Verificação de integridade de dados:** Ao aplicar uma função *hash* ao conteúdo de um conjunto de dados, é possível construir uma assinatura que atesta com grande probabilidade a integridade desses dados. De posse desta assinatura e da função utilizada, é possível verificar se os dados sofreram alguma alteração apenas aplicando a função novamente sobre os dados e verificando se as assinaturas são iguais.

- **Autenticação de mensagens:** Para autenticar mensagens, é necessário não apenas verificar sua integridade (como no exemplo anterior), mas também a autenticidade de quem criou a assinatura. Para isso utilizam-se *HMACs* (Códigos Autenticadores de Mensagens Baseados em *Hash*, em inglês), que consistem em funções *hash* que possuem dois mecanismos diferentes para geração e verificação dos *hashes*. A geração é secreta, e garante a autenticidade. A verificação é pública e permite que qualquer observador verifique a autenticidade de uma mensagem. Geralmente *HMACs* são baseados em algoritmos de criptografia assimétrica.

- **Prova de esforço:** Alguns sistemas, como forma de controle de fluxo ou segurança contra ataques de negação de serviço, podem requisitar prova de que certo esforço computacional foi feito. Um exemplo notável é o algoritmo *HashCash*, utilizado pelo protocolo de criptomoeda *BitCoin*. O algoritmo consiste em quebrar parcialmente *hashes*, isto é, encontrar mensagens cujo *hash* comece com um certo número de zeros. O esforço computacional para encontrar esse *hash* cresce exponencialmente com o número de zeros requerido. A verificação do resultado, por outro lado, pode ser feita com pouco esforço computacional.

Dependendo da aplicação, outras propriedades ainda podem ser requeridas. Por exemplo, para representação de senhas, pode ser vantajoso desenvolver uma função *hash* computacionalmente custosa, para dificultar ataques de força bruta sobre senhas curtas.

É importante notar que funções *hash* criptográficas geralmente possuem desempenho consideravelmente menor que funções não-criptográficas. Portanto seu uso acaba sendo limitado a situações onde o nível de segurança justifica queda no desempenho.

1.2.3 *Hash-flooding DoS* e *Hashing* universal

A expectativa de complexidade constante para inserção e busca em tabelas *hash* baseia-se na uniformidade da distribuição na função *hash* associada e que o número de entradas na tabela seja um fator do número de elementos. Entretanto, como esta função mapeia conjuntos potencialmente infinitos em uma tabela finita, pode haver infinitas colisões no domínio da função. Um usuário mal-intencionado de um sistema poderia utilizar conhecimento sobre a função para enviar entradas cuidadosamente criadas de forma a maximizar o número de colisões na tabela.

Esta estratégia tem o potencial de elevar a complexidade de inserção e busca na tabela para $O(n)$, o que pode ocupar mais tempo de processamento do que o projetado para o sistema. Ataques de negação de serviço baseados nesta técnica são chamados *Hash-flooding DoS* [KW11].

Este tipo de ataque é mitigável utilizando funções *hash* criptográficas. Pela propriedade da *resistência a colisão*, seria computacionalmente inviável para um adversário construir um conjunto de entradas que sejam mapeadas na mesma posição da tabela.

Estas funções, entretanto, possuem desempenho muito abaixo de suas contrapartes não-criptográficas. Idealmente uma solução para este problema deveria manter a mesma característica de desempenho de sistemas existentes.

Uma alternativa é impedir que o adversário saiba qual função *hash* está sendo utilizada pela tabela. Para isso, é possível derivar um algoritmo para construir aleatoriamente funções *hash*. Assim, cada tabela *hash* seria inicializada com uma função *hash* distinta e, sem acesso a esta função, torna-se computacionalmente inviável construir um conjunto de entradas que colidam na tabela. Este conjunto de funções é chamado de família de funções *hash* e a técnica de construção aleatória é chamada *hashing* universal.

Em muitas funções utilizadas na prática, a implementação aceita uma *semente* que inicializa a função de forma única. É importante notar que através de criptoanálise, pode-se encontrar vulnerabilidade a esquemas de colisão igualmente efetivos em todas as funções da família [BBT12]. Essas vulnerabilidades são conhecidas como *colisões universais*.

1.2.4 Exemplos de funções *hash*

Funções *hash* são amplamente utilizadas para fornecer aleatoriedade a algoritmos randomizados. Entretanto, as propriedades teóricas destes algoritmos assumem funções perfeitamente uniformes, entre outras propriedades já citadas. É inviável, na prática, encontrar tais funções. Na prática, cada função possui vantagens e desvantagens. Há situações, por exemplo, onde uma função com menor resistência a colisão pode ser usada se o ganho em desempenho justificar a escolha.

Como este é um assunto de grande aplicabilidade prática, muitas funções amplamente utilizadas na indústria possuem apenas evidência empírica de suas propriedades, o que torna difícil uma análise teórica aprofundada do estado da arte de funções *hash*.

Nesta seção discutiremos alguns exemplos de funções *hash*, sua história e propriedades. Todas as funções apresentadas mapeiam sequências de bytes ou caracteres *Unicode* para um número fixo de bits, que pode variar conforme a aplicação.

Java *string hash*

Uma das funções mais simples que podemos apresentar é a utilizada no *hashing* de *strings* em Java. De uma forma geral, dada uma sequência $s = s_1 s_2 \cdots s_n$, a função consiste em computar:

$$h(s) = \left(\sum_{i=1}^{n} 31^{n-i} s_i \right) \mod 2^{32}$$

Esta função é uma instância fixa da família de *hashes* polinomiais introduzida por Carter e Wegman [CW77]. A escolha da constante 31 parece arbitrária, mas ajuda a evitar colisões com tamanhos comuns de tabelas *hash* (comumente potências de dois), além de permitir substituir a multiplicação por um deslocamento de bits e uma subtração,

melhorando o desempenho do algoritmo.

Algoritmo 1 Computa a função *hash* do Java

Input: s: *array de bytes* $= s_1 s_2 \cdots s_n$
Output: um número de 32 bits
1: **função** JavaStringHash(s)
2: $h \leftarrow 0$
3: **para** $i \leftarrow 1$ to n **faça**
4: $h \leftarrow (31 * h + s_i) \mod 2^{32}$
5: **fim para**
6: **retorna** h
7: **fim função**

A função também pode ser generalizada como uma implementação da técnica *iterated hashing* [Lem12], que consiste na escolha de uma constante H_0 (semente) e uma função de compressão F, de forma que $H_i = F(H_{i-1}, s_i)$ para todo $i > 1$, sendo assim a função $h(s) = H_n$. No caso da função *hash* das *strings* em Java, $H_0 = 0$ e $F(H_{i-1}, s_i) = (31H_{i-1} + s_i) \mod 2^{32}$.

Não é difícil perceber que esta função não possui as características de segurança descritas na Seção 1.2.2, sendo suscetível a inversão. Por exemplo, as *strings* "BBB", "BAa"e "AaB", codificadas em ASCII, possuem todas o mesmo valor de *hash*: 65538.

Jenkins' *One-At-A-Time hashing*

A função *One-At-A-Time* é um tipo de *hash* não-criptográfico e foi desenvolvida pelo cientista da computação Bob Jenkins em 1997 [Jen97], com o objetivo de sanar problemas comuns em funções *hash* conhecidas na época. O Algoritmo 2 mostra a ideia geral da função. No algoritmo, as funções ShiftLeft(h, n) e ShiftRight(h, n) representam o deslocamento de n bits à esquerda e à direita, respectivamente. Todas as operações são feitas sobre inteiros *unsigned* de 32 bits.

O nome *One-At-A-Time* foi escolhido pois a função processa a entrada um byte por vez, em contraste com outras funções apresentadas no mesmo artigo, que processam blocos de bytes. A função possui uma formulação bastante prática com o objetivo de alcançar alto desempenho gerando *hashes* de 32 bits a cada vez.

Novas variantes deste algoritmo foram desenvolvidas por Jenkins após o artigo original. Elas buscam alcançar melhor desempenho em aplicações reais. Entretanto, todas baseiam-se na mesma ideia original. Em especial, a função mais conhecida como *Jenkinshash*, baseia-se na ideia de processar 12 bytes por vez, em vez de apenas um, otimizando o algoritmo em processadores com *pipelines* mais longas.

Algoritmo 2 Computa a função *hash One-At-A-Time*

Input: s: *array de bytes* $= s_1 s_2 \cdots s_n$
Output: um número de 32 bits

```
1: função OneAtATime(s)
2:     h ← 0
3:     para i ← 1 to n faça
4:         h ← h + s_i
5:         h ← h + ShiftLeft(h, 10)
6:         h ← h ⊕ ShiftRight(h, 6)
7:     fim para
8:     h ← h + ShiftLeft(h, 3)
9:     h ← h ⊕ ShiftRight(h, 11)
10:    h ← h + ShiftLeft(h, 15)
11:    retorna h
12: fim função
```

MurmurHash

MurmurHash é uma família de funções *hash* não-criptográficas criadas por Austin Appleby a partir de 2008. Sua grande vantagem é ter um desempenho muito superior ao das funções conhecidas na época, além de passar em todos os testes comuns de uniformidade de funções hash. O Algoritmo 3 mostra uma versão específica desta função: MurmurHash3 gerando *hashes* de 32 bits.

O nome MurmurHash é baseado nas operações que o algoritmo executa em seu *loop* principal, de *multiplicação* (MU) e *rotação* (R).

Por sua razoável uniformidade, esta função é utilizada em todos os testes experimentais ao longo deste trabalho. Porém é importante notar que esta função possui vulnerabilidades que permitem a construção de mensagens que induzem a colisão em tabelas *hash*, mesmo quando uma versão randomizada do algoritmo é utilizada, como apontado por Bernstein at al. [BBT12]. Os autores recomendam o uso de funções mais modernas, como *SipHash*.

SHA-1

SHA-1 (*Secure Hash Algorithm* 1) é uma função *hash* criptográfica desenvolvida pela NSA e publicada pelo NIST [FIP12], ambos órgãos do governo dos Estados Unidos da América, com o objetivo de se tornar o padrão de função criptográfica para assinaturas em certificados e outros fins de segurança.

A função gera *hashes* de 160 bits, geralmente representados como uma sequência hexadecimal de 40 caracteres, porém também é comum vê-los representados como *strings* codificadas em *base64* (esquema de codificação que representa cada 24 bits usando quatro caracteres), com 28 caracteres. Por exemplo, ambas as *strings*

`2fd4e1c67a2d28fced849ee1bb76e7391b93eb12` (em hexadecimal) e

Algoritmo 3 Computa a função MurmurHash3

Input: s: *array de bytes* $= s_1 s_2 \cdots s_n$
Output: um número de 32 bits

função MurmurHash3_32(s)
    $c1 \leftarrow$ `0xcc9e2d51`
    $c2 \leftarrow$ `0x1b873593`
    $h \leftarrow 0$
    para $k \leftarrow$ cada bloco completo de 4 bytes em s **faça**
        $k \leftarrow \text{RotateLeft}(k \times c1, 15) \times c2$
        $h \leftarrow h \oplus k$
        $h \leftarrow \text{RotateLeft}(h, 13)$
        $h \leftarrow h \times 5 +$ `0xe6546b64`
    fim para
    se há bloco remanscente no final de s **então**
        $k \leftarrow$ bloco remanescente no final de s
        $k \leftarrow \text{RotateLeft}(k \times c1, 15) \times c2$
        $h \leftarrow h \oplus k$
    fim se
    $h \leftarrow h \oplus n$
    $h \leftarrow h \oplus \text{ShiftRight}(h, 16)$
    $h \leftarrow h \times$ `0x85ebca6b`
    $h \leftarrow h \oplus \text{ShiftRight}(h, 13)$
    $h \leftarrow h \times$ `0xc2b2ae35`
    $h \leftarrow h \oplus \text{ShiftRight}(h, 16)$
    retorna h
fim função

`L9ThxnotKPzthJ7hu3bnORuT6xI=` (em *base64*)

representam o mesmo valor *hash*.

Entre as aplicações da função SHA-1, pode-se citar todos os protocolos baseados em SSL/TLS, incluíndo HTTPS. SHA-1 tornou-se a opção padrão para função criptográfica desde que falhas sérias de resistência a colisão foram descobertas na até então ubíqua função MD5 [WY05].

Por gerar *hashes* de 160 bits, espera-se que a função possua uma resistência a pré-imagem equivalente. Isto é, espera-se que seja necessário da ordem de 2^{160} avaliações da função para encontrar uma mensagem que corresponda a um certo valor *hash* dado. Para encontrar quaisquer duas mensagens que possuam o mesmo *hash*, espera-se da ordem de apenas 2^{80} tentativas (resistência a colisão), valendo-se do *paradoxo do aniversário*. Entretanto, em 2015, Stevens, Karpman e Peyrin [SKP15] publicaram artigo demonstrando colisões na função de compressão efetuando apenas 2^{57} avaliações. Este acontecimento diminuiu consideravelmente a confiança na função SHA-1, que já sofria ataques desde 2005.

As desenvolvedoras de *browsers* Microsoft, Google e Mozilla já anunciaram que deixa-

ram de aceitar certificados assinados com SHA-1 a partir de 2017.

Novas famílias de funções foram certificadas pelo NIST e são atualmente recomendadas para aplicações que necessitam de alto nível de segurança: SHA2, família que inclui, entre outras as funções SHA256 e SHA512; e SHA3, que é baseada na família Keccak de funções esponja (uma generalização de funções *hash*).

2 ESTRUTURAS DE DADOS PROBABILÍSTICAS

Estruturas de dados probabilísticas permitem estimar propriedades sobre fluxos de dados utilizando menos recursos que suas alternativas determinísticas. Isso é possível através da sub-representação dos dados subjacentes, para compor as operações sobre a estrutura.

Neste capítulo analisaremos quatro estruturas de dados probabilísticas que utilizam funções *hash* para representar operações em conjuntos e multiconjuntos utilizando consideravelmente menos memória que suas representações determinísticas. A Tabela 1 sumariza estas estruturas e as operações que cada uma é capaz de efetuar. Considere n a cardinalidade de elementos distintos destes (multi)conjuntos. Os detalhes de funcionamento de cada uma destas estruturas serão abordados nas próximas seções.

Tabela 1: Sumário de estruturas abordadas neste capítulo

Estrutura	**Consulta**
Filtro de Bloom	Pertinência
	Cardinalidade
Count-Min sketch	Frequência
	Produto escalar
	Somatório de intervalos
MinHash	Similaridade
HyperLogLog	Cardinalidade

2.1 Filtro de Bloom

Um filtro de Bloom é uma estrutura de dados probabilística que representa um conjunto e permite verificar a pertinência de elementos com tolerância a falsos positivos [Blo70]. É uma representação bastante compacta: são necessários menos de 10 bits por elemento para uma probabilidade 1% de falsos positivos [BMP$^+$06].

2.1.1 Definição

Um filtro de Bloom representa um conjunto S de cardinalidade n utilizando um vetor B de m bits. Inicialmente, $B[i] = 0$ para todo $i \in [1..m]$. O filtro está associado a k funções *hash* $h_i : S \to [1..m]$, para todo $i \in [1..k]$. Assume-se que cada função h_i mapeia elementos de S naqueles de $[1..m]$ de forma aleatória com uma distribuição uniforme.

Para inserir um elemento, é preciso atribuir 1 para cada posição no filtro mapeada pelas funções h_i, como mostra o Algoritmo 4.

Algoritmo 4 Adiciona um elemento a um filtro de Bloom

```
1: procedimento INSERIR(x)
2:     para i ← 1 to k faça
3:         B[h_i(x)] ← 1
4:     fim para
5: fim procedimento
```

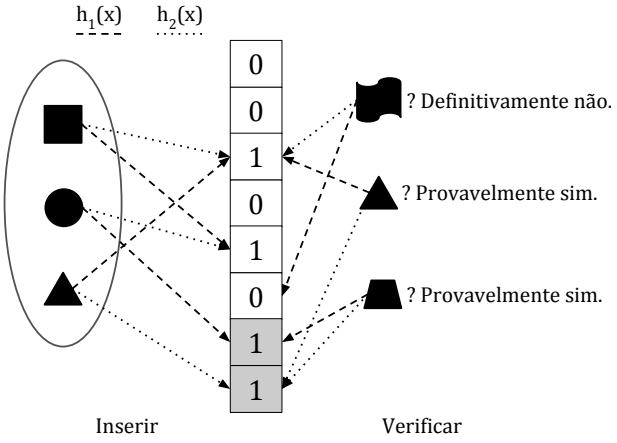
Analogamente, para verificar se um elemento pertence a um conjunto, é preciso verificar se todos os bits mapeados pelas funções h_i estão "ligados", i.e., um elemento x provavelmente pertence ao conjunto S se o Algoritmo 5 retorna verdadeiro.

Algoritmo 5 Verifica se um elemento pertence a um filtro de Bloom

```
1: função VERIFICAR(x)
2:     resultado ← true
3:     para i ← 1 to k faça
4:         resultado ← resultado ∧ B[h_i(x)] = 1
5:     fim para
6:     retorna resultado
7: fim função
```

A Figura 2 mostra um exemplo de filtro de Bloom. Do lado esquerdo, estão representadas as operações de inserção, utilizando duas funções *hash*. Do lado direito, as operações de verificação. Importante notar que na última operação de verificação trata-se de um caso de falso positivo, pois o elemento não existe de fato no conjunto previamente inserido.

Figura 2: Exemplo de filtro de bloom



Quando $k = 1$, o funcionamento de um filtro de Bloom é similar ao de uma *hash table* tradicional, porém os elementos originais não são armazenados. Em vez disso, apenas um bit é usado para representar a pertinência do *hash* do elemento na tabela. Na prática, múltiplas funções *hash* são utilizadas.

Assim como em tabelas *hash*, pode haver colisões nos filtros de Bloom. Portanto, falsos positivos são possíveis. Falsos negativos por sua vez não são possíveis. Este comportamento é desejado em sistemas que precisam evitar operações custosas (por exemplo, acesso a disco, comunicação de rede, etc.) em casos onde não seja realmente necessário.

Para diminuir a probabilidade de colisão, é preciso dimensionar o filtro e as funções *hash* baseados no número de elementos esperados. Por exemplo, usando $k = 7$ funções e $m = 10n$ bits, é possível garantir uma probabilidade de falsos positivos menor que 1% [BMP+06]. É possível determinar a quantidade ótima de bits e funções dada uma probabilidade esperada de falsos positivos.

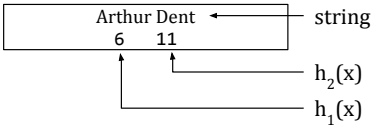
As funções *hash* usadas no filtro precisam ser independentes duas-a-duas, o que pode exigir muitos recursos computacionais. Em [KM06b], Kirsch e Mitzenmacher mostram que é possível implementar um filtro de Bloom com qualquer valor de k apenas combinando duas funções *hash* independentes, sem aumentar a probabilidade de falsos positivos.

Em termos de operações suportadas, filtros de Bloom permitem atualização incremental, bem como união entre filtros (que para filtros de mesmo número de bits, consiste em uma operação OU bit-a-bit entre eles). Entretanto, em sua forma mais simples, não suportam remoção de elementos nem interseção entre filtros sem introduzir falsos negativos ou afetar as probabilidades de falsos positivos. Diversas variações e extensões aos filtros de Bloom foram propostas, algumas discutidas nas Seções 2.1.5 e 2.1.6.

2.1.2 Exemplo

Com o objetivo de facilitar o entendimento da estrutura filtro de Bloom, apresentamos aqui um exemplo prático de seu uso. Consideraremos aqui a simulação de inserção de strings em um filtro de Bloom com $m = 16$, usando duas funções *hash* que retornam valores no intervalo $[1; 16]$. A Figura 3 explica como será representada cada entrada no exemplo.

Figura 3: Modelo de linha para exemplos



A simulação é dividida em duas partes: inserções e consultas. Na parte das inserções, cada linha (exceto a primeira) contém uma string a ser inserida e os valores retornados por cada função *hash*. Esses valores representam os índices que serão escritos no vetor. Além disso, para cada string, são representadas as 16 posições do vetor B com o estado após a inserção, destacando os índices referenciados. A primeira linha representa o estado inicial. Na parte das consultas, algumas consultas são representadas da mesma maneira que as inserções, entretanto, o vetor não é modificado. A simulação pode ser vista na

Figura 4.

Figura 4: Exemplos de inserção e consultas no filtro de Bloom

String	B[1..m]															
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Arthur Dent 6 11	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**
Tricia McMillan 10 9	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**
Zaphod Beeblebrox 4 12	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**
Ford Prefect 3 1	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**
Marvin 14 4	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**

Consulta	B[1..m]																Resposta
Ford Prefect 3 1	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**Sim**
Slartibartfast 14 5	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**Não**
Fenchurch 11 3	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**Sim**

As consultas efetuadas representam resultados diferentes. A consulta por `Ford Prefect` retorna um resultado positivo, pois ambos os índices apontam para valores 1 na tabela, indicando que a string de fato existe no conjunto. A consulta por `Slartibartfast` retorna um resultado negativo, pois um dos índices aponta para um valor 0 na tabela, indicando que a string não existe no conjunto. Este resultado é dado com 100% de certeza. A consulta por `Fenchurch` retorna um resultado positivo, porém este é um falso positivo, visto que esta string não faz parte do conjunto.

2.1.3 Probabilidade de falso positivo

A probabilidade de um falso positivo é determinada pela probabilidade de colisão. Assim, quanto mais bits forem usados no filtro de Bloom, menor a probabilidade de um falso positivo. Para diminuir a probabilidade de colisões é possível também utilizar múltiplas funções *hash* para cada elemento, mas somente até um certo ponto, onde realizar múltiplos hashes por elemento acaba aumentando a probabilidade de colisões.

É possível calcular a probabilidade de um falso positivo. Considere uma tabela $B[1..m]$, após inserir os elementos do conjunto S, com $|S| = n$, e usando k funções *hash*. Se X_i é a variável aleatória que fornece o valor de $B[i]$, considerando a independência das funções *hash* utilizadas, então:

$$\Pr[X_i = 0] = \Pr\left[\bigwedge_{x \in S, 1 \leq j \leq k} h_j(x) \neq i\right] = \left(1 - \frac{1}{m}\right)^{kn}$$

Seja FalsoPositivo o evento do filtro reportar pertinência para um elemento $y \notin S$.

Logo, a probabilidade de observar k vezes $X_i = 1$, ou seja:

$$\Pr[\textsc{FalsoPositivo}] = \Pr\left[\bigwedge_{1 \le j \le k} X_{h_j(y)} = 1\right] = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - \mathrm{e}^{-kn/m}\right)^k$$

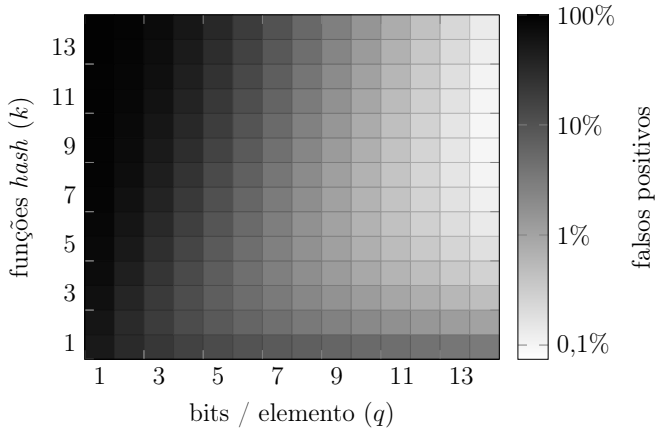
A aproximação acima se dá pois $1 - x \approx \mathrm{e}^{-x}$, conforme x se aproxima de zero. É fácil ver que a probabilidade de falsos positivos cresce conforme n cresce. Por isso, é preciso dimensionar o filtro de Bloom de acordo com o número de elementos no conjunto. Se definirmos $q = m/n$ (quantidade de bits por elemento, reescrevemos a última probabilidade como:

$$\Pr[\textsc{FalsoPositivo}] \approx \left(1 - \mathrm{e}^{-k/q}\right)^k$$

A Figura 5 mostra como esta probabilidade se comporta para diferentes valores de k e q. Bonomi et al. [BMP+06] mostram ainda que a probabilidade é minimizada quando $k = q \ln 2$. Assim, é possível definir a probabilidade mínima de falsos positivos a partir do número de bits por elemento:

$$\Pr[\textsc{FalsoPositivo}] = (1 - e^{-\ln(2)})^{q \ln(2)} = ((1/2)^{\ln(2)})^q \approx (0,6185)^q$$

Figura 5: Probabilidade de falsos positivos

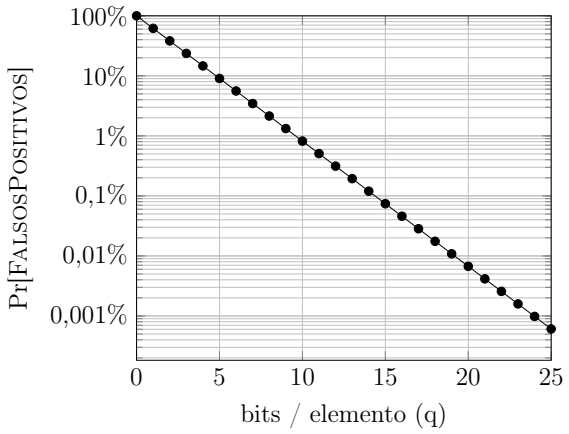


É possível ver na Figura 6 que a probabilidade de falsos positivos cai exponencialmente em relação ao número de bits por elemento no filtro.

Desta forma, a escolha do número de bits por elemento torna-se crucial para dimensionar a quantidade de falsos positivos. Esta análise precisa ser feita *a priori*, pois não é possível redimensionar um filtro de Bloom sem modificar suas propriedades estatísticas.

Existem variações do algoritmo que mitigam este problema. Por exemplo, os *Dynamic Bloom Filters* [GWC+10] colocam um limite superior na probabilidade de falso positivo criando um novo filtro quando o limite é ultrapassado, penalizando as consultas em um fator logaritmico, em relação ao tamanho do conjunto, pois precisam verificar vários fil-

Figura 6: Probabilidade mínima de falsos positivos



tros. Já os *Block-partitioned Bloom Filters* [PSN10] penalizam tanto a inserção quanto a consulta de forma similar, porém por inserir e verificar os elementos em todos os filtros, mantêm as propriedades algébricas de união entre filtros através de operações booleanas entre seus vetores.

2.1.4 Estimativa de cardinalidade

É possível estimar a cardinalidade do conjunto representado por um filtro de Bloom [WVZT90, PSN10]. O princípio é análogo àquele empregado na seção anterior para estimar a probabilidade de falsos positivos. Seja T a variável aleatória que representa o número de bits 1 após inserir n elementos num filtro $B[1..m]$ e k funções *hash*. Portanto, $T = \sum_{1 \leq i \leq m} X_i = 1$ e a esperança $E[T]$ é uma função $S(n)$ (para m e k fixos) tal que:

$$S(n) = E[T] = \sum_{1 \leq i \leq m} E[X_i] = m \times \Pr[X_i = 1] = m \times \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)$$

Desta forma, é possível estimar n a partir do número t de bits 1 existentes no filtro:

$$n = \hat{S}^{-1}(t) = \frac{\ln\left(1 - \frac{t}{m}\right)}{k \times \ln\left(1 - \frac{1}{m}\right)} \approx \frac{-m}{k} \times \ln\left(1 - \frac{t}{m}\right)$$

A aproximação acima se dá pois $\ln(1 + x) \approx x$, conforme x se aproxima de zero. Perceba que esta estimativa é extremamente dependente da quantidade de bits por elemento no filtro. Portanto, dado um certo filtro de Bloom, apenas um intervalo definido de cardinalidades tem um erro dentro de um limite aceitável. Papapetrou et al. [PSN10] mostram que é possível definir um limite inferior na probabilidade da cardinalidade real estar num certo intervalo (n_a, n_b) (com $S^{-1}(t-1) \leq n_a \leq n_b \leq S^{-1}(t+1)$). Esta probabilidade se

dá pela fórmula:

$$\Pr[n_a \leq n \leq n_b] \geq 1 - e^{-\frac{(t+1-S(n_b))^2}{2S(n_b)}} - e^{t-1-S(n_a)} \times (S(n_a)/(t-1))^{t-1}$$

Este é apenas um limite inferior. Na prática é possível estimar valores bem maiores. Em especial, para estimar cardinalidades não há vantagens em utilizar múltiplas funções *hash*, pois isto somente aumentaria a densidade de bits iguais a 1 no filtro. Whang et al. introduziram o algoritmo Linear Counting [WVZT90] que utiliza um filtro de Bloom com apenas uma função *hash* para estimar a cardinalidade. Assim,

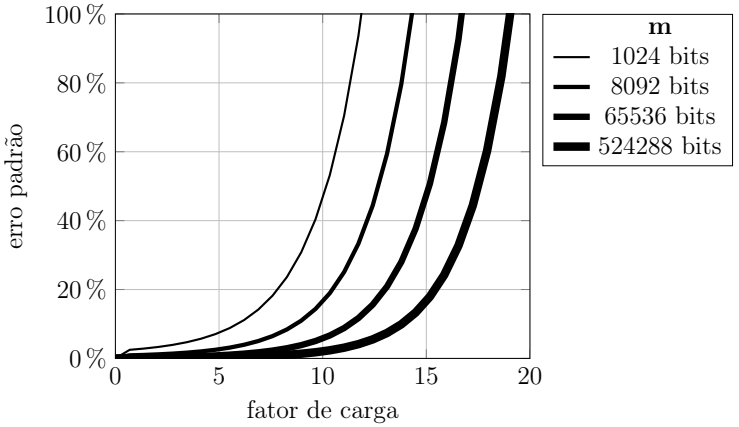
$$\hat{n} \approx -m \times \ln\left(1 - \frac{t}{m}\right)$$

Em [WVZT90] também é mostrado que o erro padrão da estimativa está fortemente atrelada ao fator de carga, que consiste em quantos elementos distintos foram inseridos para cada bit na estrutura, ou n/m, expresso a seguir:

$$\sigma_{\hat{n}} = \frac{\sqrt{m(e^{n/m} - (n/m) - 1)}}{n}$$

Assim, assumindo um filtro com quantidades diversas de bits (m), é possível ver a degradação do erro padrão conforme o fator de carga aumenta (Figura 7).

Figura 7: Erro padrão por fator de carga para filtros de vários tamanhos.



2.1.5 *Counting Bloom filters*

Em sua forma mais simples, o filtro de Bloom não permite remoção de elementos. Uma solução trivial para este problema, introduzida por Fan et al. [FCAB98], é manter um contador de b bits para cada posição no vetor B do filtro.

Desta forma, as operações originais do filtro de Bloom são estendidas. A inserção passa a incrementar o valor de cada uma das posições resultantes das funções *hash* (Algoritmo 6).

A remoção (Algoritmo 7) e verificação (Algoritmo 8) são análogas.

Algoritmo 6 Adiciona um elemento a um *Counting Bloom Filter*

```
1: procedimento INSERIR(x)
2:     para i ← 1 to k faça
3:         se B[h_i(x)] < 2^b − 1 então
4:             B[h_i(x)] ← B[h_i(x)] + 1
5:         fim se
6:     fim para
7: fim procedimento
```

Algoritmo 7 Remove um elemento de um *Counting Bloom Filter*

```
1: procedimento REMOVER(x)
2:     para i ← 1 to k faça
3:         se B[h_i(x)] < 2^b − 1 então
4:             B[h_i(x)] ← B[h_i(x)] − 1
5:         fim se
6:     fim para
7: fim procedimento
```

Algoritmo 8 Verifica se um elemento pertence a um *Counting Bloom Filter*

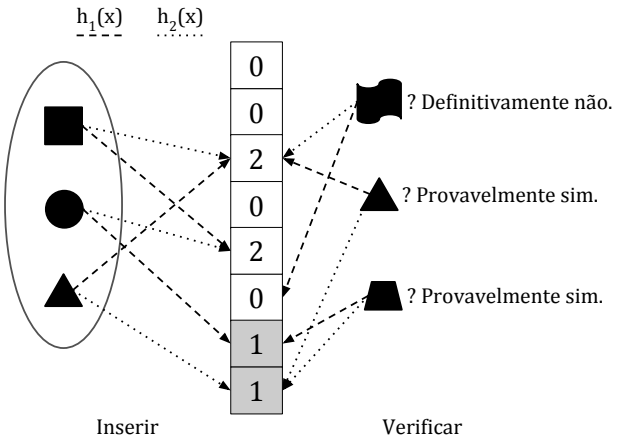
```
1: função VERIFICAR(x)
2:     resultado ← true
3:     para i ← 1 to k faça
4:         resultado ← resultado ∧ B[h_i(x)] ≥ 1
5:     fim para
6:     retorna resultado
7: fim função
```

Agora a Figura 8 mostra um exemplo de filtro de Bloom com contagem. Do lado esquerdo figuram as operações de inserção, com duas funções *hash*. Perceba que colisões incrementam a posição resultante no vetor. Do lado direito estão as verificações. Na terceira verificação nota-se um falso positivo.

A remoção neste filtro baseia-se na ausência de *overflows* (quando um valor do vetor B atinge 2^b) em seus contadores. Para tanto, é preciso dimensionar o tamanho destes contadores de forma a minimizar a probabilidade de overflow. Como originalmente o filtro usa apenas um bit por posição no vetor, qualquer número de bits adicionais escolhidos para representar o contador aumenta significativamente o espaço necessário para armazenar a estrutura. Normalmente, 4 bits são suficientes para a maior parte das aplicações, o que faz com que estes filtros usem quatro vezes mais espaço que filtros normais.

O número de bits por contador determina a probabilidade de *overflow*. *Overflows* em *counting Bloom filters* precisam ser tratados. Se um *overflow* ocorrer é preciso manter

Figura 8: Exemplo de filtro de bloom com contagem



o contador no máximo valor possível. De forma equivalente, ao tentar decrementar uma posição já no maior valor possível, é preciso mantê-la nesse mesmo valor. Faz-se isto para evitar a introdução de falsos negativos.

Na prática, entretanto, *overflows* devem ser consideravelmente raros. A probabilidade de um *overflow* dado que cada contador usa b bits, após inserir n elementos, usando $m = 10n$ contadores e otimizando o número de funções *hash* é:

$$\Pr[\text{OVERFLOW}] \leq n \left(\frac{e \ln 2}{2^b} \right)^{2^b}$$

como provado por Fan et al. [FCAB98]. Assim, ao usar 4 bits para os contadores, a probabilidade de overflow concentra-se em:

$$\Pr[\text{OVERFLOW EM 4 BITS}] \leq 1,37 \times 10^{-15} \times n$$

Esta probabilidade baseia-se na possibilidade de produzir funções *hash* realmente aleatórias, o que pode ser aproximado, mas não pode ser garantido. Em um pior caso em especial, se o mesmo elemento for repetidamente oferecido à estrutura, a probabilidade de *overflow* torna-se criticamente alta.

2.1.6 Outras variantes

O filtro de Bloom, em sua simplicidade, tem limitações que posam como desafios para algumas aplicações. Desde sua concepção, entretanto, muitas extensões foram propostas para superar estas limitações (geralmente em detrimento de algum parâmetro de desempenho), entre elas:

***Compressed Bloom Filters* [Mit02]** Discute esquemas de compressão para otimizar o tamanho de filtros enviados através de rede ou salvos em disco. O objetivo é

construir filtros com mais bits e menos funções *hash*, de forma a minimizar o tamanho comprimido e/ou aumentar a precisão. Especialmente útil em caso de caches distribuídos.

***Distance-sensitive Bloom Filters* [KM06a]** Responde consultas do tipo "algum elemento próximo a x pertence a S?", dada uma métrica apropriada, utilizando funções *hash* sensíveis a localidade, com possibilidade tanto de falsos positivos como falsos negativos. Esta variante é bastante útil para detecção de plágio, por exemplo.

***Dynamic Bloom Filters* [GWC$^+$10]** Permite redimensionar dinamicamente filtros de Bloom sem perder suas propriedades estatísticas. Baseia-se na criação de múltiplos filtros com um limite superior na probabilidade de falsos positivos, i.e., quando um filtro alcança uma taxa de falsos positivos muito alta, outro filtro vazio é criado. Nesta modalidade, a complexidade das operações padrão é maior, entretanto, na prática isso não posa como um problema. É possível dimensionar o tamanho dos filtros a serem criados de forma a amortizar a quantidade de filtros criados ao longo do tempo.

***Spectral Bloom Filters* [CM03]** É uma estrutura especializada em lidar com multiconjuntos. Um *Spectral Bloom Filter* (SBF) funciona praticamente da mesma forma que um *Counting Bloom Filter*, mas suas operações são especializadas em obter a multiplicidade de um certo item num multiconjunto. Nesta estrutura, a multiplicidade de um elemento x se dá por $\min_{1 \leq i \leq k} B[h_i(x)]$. A estrutura SBF é conceitualmente similar à *Count-Min*, apresentada na Seção 2.2.

2.1.7 Aplicações

Filtros de Bloom são estruturas simples, porém têm aplicabilidade em muitos domínios diferentes. Eles são especialmente importantes em sistemas que desejam diminuir o custo de verificar se uma operação mais custosa precisa ser feita (como a de determinar se um arquivo está armazenado antes de recorrer ao disco). Estes sistemas estão preparados para lidar com alguns falsos positivos, mas beneficiam-se ao não precisar efetuar tais operações em caso negativo.

Processadores de texto: O artigo original sobre os filtros de Bloom propõe uma aplicação que avalia regras de hifenização [Blo70]. Segundo Bloom, no caso descrito, 90% das regras de hifenização de palavras poderiam ser generalizadas por regras simples e apenas 10% iriam requerer uma consulta ao disco. Neste caso, um filtro de Bloom seria utilizado para verificar se a palavra é uma das que estão no disco. O falso positivo somente causaria uma ida desnecessária ao disco, o que ainda seria uma vantagem, já que a maioria dos casos poderia ser resolvida em memória.

O mesmo princípio pode ser aplicado para verificação ortográfica. Ramakrishna [Ram89] discute como utilizar filtros de Bloom pode diminuir o espaço necessário para armazenar vários dicionários ao mesmo tempo em memória.

Bancos de dados: Há diversas aplicações para filtros de Bloom em bancos de dados. Mullin [Mul93] descreve um método para estimar o resultado de junções (*joins*) relacionais distribuídos. O filtro é bastante apropriado em sistemas distribuídos, pois diminui a necessidade de comunicação entre os nós para computar alguns resultados. Antognini [AT08] mostra usos dos filtros de Bloom no banco de dados Oracle tanto para computar *joins* distribuídos quanto para cache de resultados.

Controle de tráfego de redes: Feng et al. [FKSS99] descrevem uma classe de algoritmos para controle de tráfego de redes conhecido como BLUE. Uma das variantes deste algoritmo, conhecida como STOCHASTIC FAIR BLUE (SFB), estimula um controle de tráfego que pune hosts que congestionam a rede. Muitas vezes o custo de espaço para manter informações sobre esses hosts em memória pode ser impraticável, principalmente considerando a quantidade limitada de recursos em equipamentos de redes. O algoritmo SFB utiliza então um filtro de Bloom para manter estas informações.

Caches distribuídos: Em sistemas de cache distribuído é importante que cada nó do cluster possa saber quais chaves seus vizinhos possuem. Uma das técnicas frequentemente empregadas são os *cache digests*, que são uma forma de compressão com perda de todas as chaves presentes em um nó. *Cache digests* usam, entre outras coisas, filtros de Bloom [RW98]. Periodicamente os nós trocam *cache digests* entre si para que o conhecimento sobre quais chaves estão em cada um de seus vizinhos seja disseminado. Neste caso, o custo de um falso positivo somente implica em uma requisição a mais para verificar se de fato a chave existe.

Verificação de URLs maliciosas: O navegador Google Chrome usa filtros de Bloom para verificar se a *URL* digitada pelo usuário faz parte do banco de dados de sites maliciosos [Hon06]. Assim, casos negativos são rapidamente verificados. E na minoria dos casos, quando há um falso positivo, basta uma requisição de rede a mais para retificar a informação.

2.1.8 Resultados experimentais

Para melhor observar as previsões teóricas sobre os filtros de Bloom, conduzimos uma série de experimentos com dados reais e sintéticos para verificar empiricamente as probabilidades descritas na teoria. Testamos duas variantes: teste de pertinência com filtro de Bloom clássico e estimativa de cardinalidade com LINEAR COUNTING [WVZT90] .

Foram utilizados dois conjuntos de dados: um composto por todas as palavras nas obras de Shakespeare (964410 palavras, 23704 distintas) e outro composto por cadeias aleatórias de 32 caracteres (1064960 cadeias no total).

Em todos os testes, a função *hash* utilizada foi MurmurHash 3, de 32 bits.

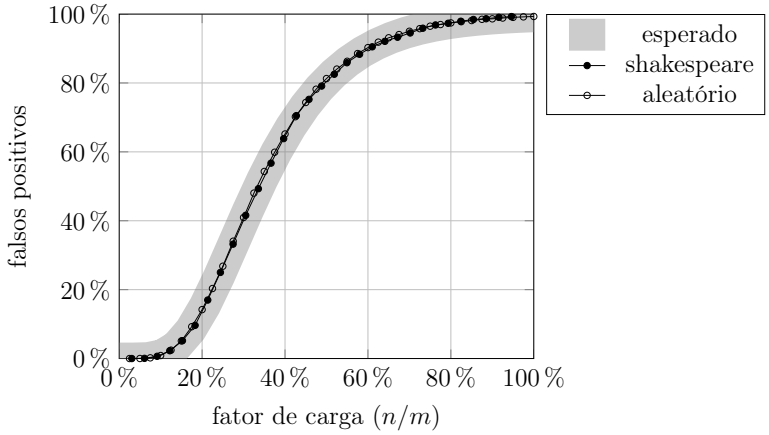
Com estes dados, foram realizados dois testes: um para observar a quantidade de falsos positivos e outro para observar o desvio entre a cardinalidade estimada e a real. Os filtros de Bloom foram dimensionados de acordo com cada conjunto de dados, como descrito na Tabela 2. Para ambos os conjuntos de dados, foram utilizadas 7 funções *hash* no filtro para falsos positivos.

Tabela 2: Configuração dos valores de m nos filtros de teste.

	Shakespeare	**Cadeias aleatórias**
filtro para testar falsos positivos	16384 bits	1048576 bits
filtro para testar cardinalidade	2048 bits	131072 bits
palavras inseridas	20000	1048576
palavras verificadas	3704	16384

Durante a inserção das palavras, a cada 2,5% de palavras inseridas, o teste era realizado: no caso dos falsos positivos, o conjunto de verificação era executado contra o filtro verificando a porcentagem de falsos positivos; no caso do teste de cardinalidade, estimando a cardinalidade e registrando a razão entre esta e a cardinalidade real do conjunto até o momento. Como os dois testes foram realizados com conjuntos de tamanhos diferentes, todos os resultados serão apresentados aqui em valores relativos de fator de carga. No teste de falsos positivos, como pode ser visto na Figura 9, a quantidade dos mesmos ficou extremamente aderente à teoria. É possível perceber como o fator de carga sozinho é capaz de influenciar a probabilidade de falsos positivos. O valor esperado neste teste é o mesmo descrito na Seção 2.1.3.

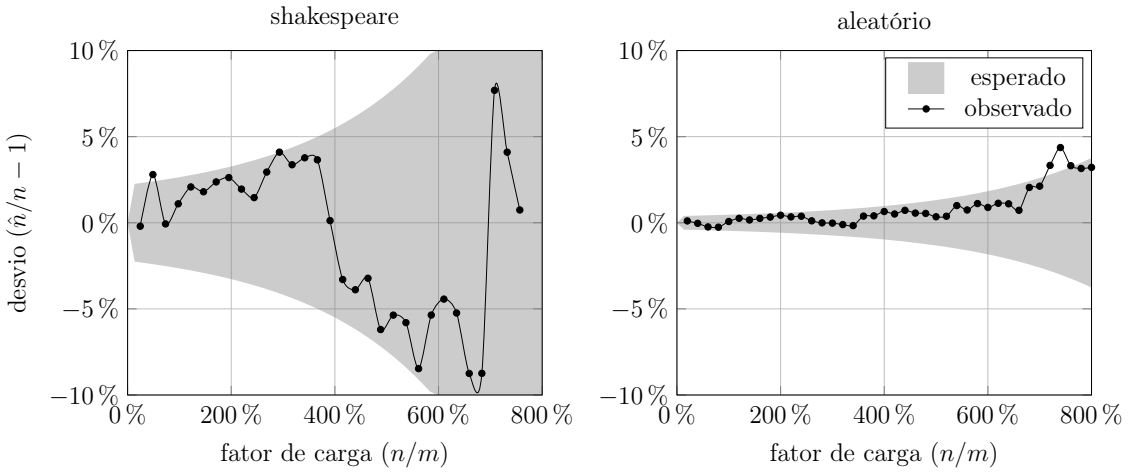
Figura 9: Falsos positivos observados e probabilidade esperada.



Para o teste de cardinalidade foram utilizados filtros com uma quantidade de bits

menor, para poder observar fatores de carga maiores que 100%. O resultado pode ser visto na Figura 10. É possível perceber que aumentando o tamanho do filtro (m) a precisão aumenta, mesmo quando comparado em termos relativos de fator de carga.

Figura 10: Desvio observado por fator de carga



Na figura, o erro esperado é apresentado no valor de 2 erros padrão, desta forma, espera-se que aproximadamente 95% das estimativas estejam dentro deste erro.

Apesar do teste apresentar uma flutuação maior no resultado, observa-se com clareza como a estimativa de cardinalidades continua com resultados satisfatórios (erro menor que 10%) para fatores de carga muito maiores que 100%.

2.2 *Count-Min Sketch*

Count-Min Sketch é uma estrutura probabilística, descrita por Cormode e Muthukrishnan [CM05], que permite representar um vetor implicitamente e estimar consultas sobre ele. Esta estrutura mostra-se importante em casos onde o vetor representado não caberia em memória, sendo aceitáveis resultados probabilísticos para consultas específicas. Em especial, *Count-Min Sketch* e suas variantes permitem estimar o valor em índices e intervalos específicos do vetor, bem como o produto escalar entre diferentes vetores. Estas operações ajudam a resolver muitos problemas relacionados à análise de fluxos de dados. Em especial, ao representar os elementos de um multiconjunto de inteiros como os índices do vetor A, e suas frequências como os valores, é possível utilizar a consulta de intervalo para estimar os percentis do multiconjunto original.

2.2.1 Definição

O objetivo de *Count-Min Sketch* é representar um vetor A, definido incrementalmente através de operações de atualização na forma de pares ordenados (i, c), que representam um acréscimo de c unidades na i-ésima posição do vetor, isto é $A[i] \leftarrow A[i] + c$. Uma

outra forma de ver esta operação é como um acréscimo de c unidades na multiplicidade do elemento i em um multiconjunto representado pelo vetor A. O vetor e suas atualizações não precisam ser mantidos em memória. A estrutura permite a qualquer momento, efetuar consultas sobre parâmetros do vetor original, respondidas probabilisticamente. Em especial, descreveremos nesta seção o processo para responder as seguintes consultas:

- Para um índice i, o valor $A[i]$;
- Para vetores A e B, o produto escalar $A \cdot B$.

Count-Min Sketch funciona de forma similar a um filtro de Bloom com contagem. Entretanto, em vez de apenas um vetor, a estrutura utiliza uma matriz $M[1..k, 1..m]$, onde cada linha corresponde a uma função *hash*.

A atualização parte de um par ordenado (i, c), representando um incremento de valor c na i-ésima posição do vetor implícito A. O algoritmo consiste em incrementar na matriz todas as posições referenciadas pelos hashes calculados em cada um dos seus respectivos vetores (Algoritmo 9).

Algoritmo 9 Atualiza Count-Min

```
1: procedimento ATUALIZAR(i, c)
2:     para j ← 1 to k faça
3:         M[h_j(i), j] ← M[h_j(i), j] + c
4:     fim para
5: fim procedimento
```

A consulta por valor se dá obtendo o mínimo entre todas as células referenciadas pelas funções *hash* (Algoritmo 10).

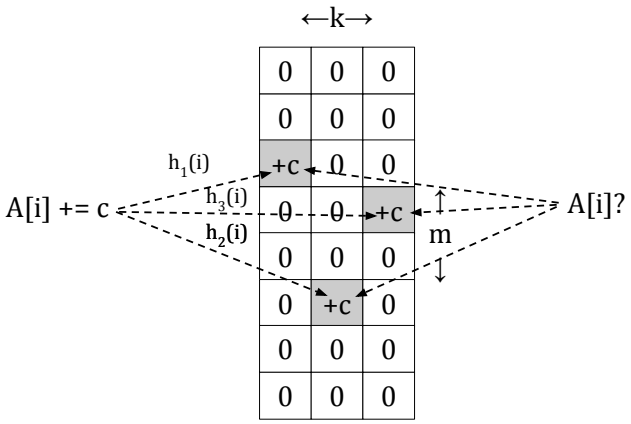
Algoritmo 10 Estima valor de $A[i]$

```
1: função ESTIMAR(i)
2:     resultado ← ∞
3:     para j ← 1 to k faça
4:         resultado ← min(resultado, M[h_j(i), j])
5:     fim para
6:     retorna resultado
7: fim função
```

A Figura 11 exemplifica os dois processos anteriores.

Percebe-se intuitivamente que o valor estimado de um elemento nunca é menor que seu valor real, isto é, $A[i] \leq \hat{A[i]}$ para todo $1 \leq i \leq n$ (assumindo que A consiste de elementos não-negativos). Este resultado é importante, pois representa muitos casos reais de uso, como contagem de acessos de usuários ou monitoramento de uso e liberação de recursos compartilhados.

Figura 11: Atualização e consulta em um *Count-Min Sketch*



Dadas duas matrizes *Count-Min* $M_A[1..k, 1..m]$ e $M_B[1..k, 1..m]$ representando, respectivamente, vetores implícitos A e B, é possível estimar o produto escalar $A \cdot B$ obtendo o mínimo valor entre os produtos escalares das respectivas linhas nas duas matrizes, como mostrado no Algoritmo 11.

Algoritmo 11 Estima $A \cdot B$

```
1: função PRODUTO-ESCALAR(M_A, M_B)
2:     resultado ← ∞
3:     para i ← 1 to k faça
4:         soma ← 0
5:         para j ← 1 to m faça
6:             soma ← soma + M_A[i, j] · M_B[i, j]
7:         fim para
8:         resultado ← min(resultado, soma)
9:     fim para
10:    retorna resultado
11: fim função
```

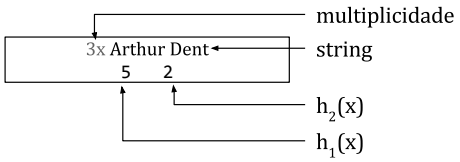
De forma análoga à consulta pontual, caso ambos os vetores tenham apenas elementos não-negativos, vale que $A \cdot B \leq \widehat{A \cdot B}$.

2.2.2 Exemplo

Com o objetivo de facilitar o entendimento da estrutura *Count-Min sketch*, apresentamos aqui um exemplo prático de seu uso. Consideraremos aqui a simulação de inserção de strings em multiplicidades variadas em um *Count-Min sketch* com $m = 8$, usando duas funções *hash* que retornam valores no intervalo $[0; 7]$. A Figura 12 explica como será representada cada entrada no exemplo.

A simulação é dividida em duas partes: inserções e consultas. Na parte das inserções, cada linha (exceto a primeira) contém uma string a ser inserida e os valores retornados por cada função *hash*. Esses valores representam os índices que serão escritos na matriz.

Figura 12: Modelo de linha para exemplos



Além disso, para cada string, são representadas as 8 colunas da matriz M com o estado após a inserção, destacando os as células referenciadas. A primeira linha representa o estado inicial. Na parte das consultas, algumas consultas são representadas da mesma maneira que as inserções, entretanto, o vetor não é modificado. A simulação pode ser vista na Figura 13.

Figura 13: Exemplos de inserção e consultas em *Count-Min sketch*

Strings inseridas	M[0..m-1]							
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
3x Arthur Dent	**0**	**0**	**0**	**0**	**0**	**3**	**0**	**0**
5 2	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**0**
2x Tricia McMillan	**0**	**2**	**0**	**0**	**0**	**3**	**0**	**0**
1 0	**2**	**0**	**3**	**0**	**0**	**0**	**0**	**0**
2x Zaphod Beeblebrox	**0**	**2**	**0**	**0**	**0**	**5**	**0**	**0**
5 3	**2**	**0**	**3**	**2**	**0**	**0**	**0**	**0**
5x Ford Prefect	**0**	**2**	**5**	**0**	**0**	**5**	**0**	**0**
2 0	**7**	**0**	**3**	**2**	**0**	**0**	**0**	**0**
1x Marvin	**0**	**2**	**5**	**0**	**0**	**6**	**0**	**0**
5 3	**7**	**0**	**3**	**3**	**0**	**0**	**0**	**0**

Consultas	M[0..m-1]								Resposta
Ford Prefect	**0**	**2**	**5**	**0**	**0**	**6**	**0**	**0**	**5**
2 0	**7**	**0**	**3**	**3**	**0**	**0**	**0**	**0**	
Marvin	**0**	**2**	**5**	**0**	**0**	**6**	**0**	**0**	**3**
5 3	**7**	**0**	**3**	**3**	**0**	**0**	**0**	**0**	
Fenchurch	**0**	**2**	**5**	**0**	**0**	**6**	**0**	**0**	**0**
1 7	**7**	**0**	**3**	**3**	**0**	**0**	**0**	**0**	

As consultas efetuadas representam resultados diferentes, definido pelo mínimo entre todas as células referenciadas. A consulta por `Ford Prefect` retorna 5, o valor exato da multiplicidade no multiconjunto. A consulta por `Marvin` retorna 3, superestimando a única ocorrência do valor no multiconjunto. A consulta por `Fenchurch` retorna 0, indicando que a string de fato não faz parte do conjunto.

2.2.3 Estimativa do erro

Em [CM05] é mostrado que a escolha dos parâmetros m e k pode ser feita em função dos parâmetros ϵ (o fator de erro) e δ (a confiança do erro). Com efeito, dados $m = \lceil e/\epsilon \rceil$ e $k = \lceil \ln(1/\delta) \rceil$, é possível mostrar que o erro da estimativa $\widehat{A[i]}$ se dá por um fator de ϵ. Mais especificamente, se A consiste de elementos não-negativos,

$$A[i] \leq \widehat{A[i]} \leq A[i] + \epsilon \|A\|_1$$

com probabilidade inferior a $1 - \delta$ somente para o limite superior, o limite inferior se dá de forma determinística.

Para demonstrar este resultado, precisamos introduzir uma variável aleatória $I_{h,i,j}$, que representa, para cada função *hash*, incidência de dois índices distintos em A na mesma linha na matriz. Isto é, para cada função h

$$I_{h,i,j} = \begin{cases} 1 & \text{se } i \neq j \wedge h(i) = h(j) \\ 0 & \text{caso contrário} \end{cases}$$

Perceba que $I_{h,i,j}$ é uma variável de Bernoulli, portanto,

$$\mathrm{E}[I_{h,i,j}] = \Pr[h(i) = h(j)] = \frac{1}{m} = \frac{\epsilon}{e},$$

assumindo que $m = e/\epsilon$.

Definimos também uma variável $X_{h,i}$, que representa, para cada função *hash*, o somatório de elementos em A (exceto o próprio índice i) que foram adicionados na mesma célula da matriz que $A[i]$, isto é

$$X_{h,i} = \sum_{j \in [1..n] \setminus \{i\}} I_{h,i,j} A[j]$$

A variável $X_{h,i}$ pode ser interpretada como o erro na estimativa para cada função *hash*, isto é $M[h_q(i), q] = A[i] + X_{h_q,i}$. Como todo $A[i]$ é não-negativo, $X_{h,i}$ também é não-negativo, portanto $\widehat{A[i]} \geq A[i]$. Além disso, pela linearidade da expectativa,

$$\mathrm{E}[X_{h,i}] = \mathrm{E}\left[\sum_{j \in [1..n] \setminus \{i\}} I_{h,i,j} A[j]\right] = \mathrm{E}[I_{h,i,j}] \sum_{j=1}^{n} A[j] = \frac{\epsilon}{e} \|A\|_1$$

onde $\|A\|_1$ representa a norma l1 do vetor, isto é, a soma de seus elementos. Para provar o limite superior, analisaremos a probabilidade de uma estimativa estar acima deste limite. Para isso, todas as funções *hash* precisam gerar estimativas acima do limite, ou seja

$$\begin{aligned} \Pr\left[\widehat{A[i]} > A[i] + \epsilon\|A\|_1\right] &= \Pr\left[\forall_{q \in [1..k]} M[h_q(i), k] > A[i] + \epsilon\|A\|_1\right] \\ &= \Pr\left[\forall_{q \in [1..k]} A[i] + X_{h_q,i} > A[i] + \epsilon\|A\|_1\right] \end{aligned}$$

e como $e\mathrm{E}[X_{h,i}] = \epsilon\|A\|_1$, podemos dizer que

$$\begin{aligned} \Pr\left[\widehat{A[i]} > A[i] + \epsilon\|A\|_1\right] &= \Pr\left[\forall_{q \in [1..k]} X_{h_q,i} > e\mathrm{E}[X_{h_q,i}]\right] \\ &= \left(\Pr\left[X_{h_q,i} > e\mathrm{E}[X_{h_q,i}]\right]\right)^k \end{aligned}$$

portanto, pela desigualdade de Markov,

$$\Pr\left[\widehat{A[i]} > A[i] + \epsilon\|A\|_1\right] < \frac{1}{e^k}$$

Assumindo que $k = \ln(1/\delta)$, podemos afirmar que

$$\Pr\left[\widehat{A[i]} > A[i] + \epsilon\|A\|_1\right] < \delta.$$

Ou seja, para o caso $\forall_i A[i] \geq 0$, a probabilidade de a estimativa $\widehat{A[i]}$ não ultrapassar o limite superior definido é maior que $1 - \delta$.

É possível utilizar uma variante do algoritmo para o caso geral, onde $A[i]$ pode assumir valores negativos no momento da estimativa. Neste caso, utiliza-se a mediana como estimativa no lugar do mínimo. É possível então demonstrar que

$$A[i] - 3\epsilon\|A\|_1 \leq \widehat{A[i]} \leq A[i] + 3\epsilon\|A\|_1$$

com probabilidade $1 - \delta^{1/4}$. Basta observar que dado que $M[h_q(i), q] = A[i] + X_{h_q,i}$, segue

$$\mathrm{E}\left[|M[h_q(i), q] - A[i]|\right] = \mathrm{E}[X_{h_q,i}] = \frac{\epsilon}{e}\|A\|_1$$

Aplicando novamente a desigualdade de Markov, mostra-se que a probabilidade de a estimativa de cada função *hash* estar errada por um valor absoluto maior que $3\epsilon\|A\|_1$ é menor que $1/3e$. Como estamos obtendo a mediana de $\lceil \ln(1/\delta) \rceil$ estimadores, a probabilidade de pelo menos metade deles estar acima do limite de erro que pretendemos provar, pelo limite de Chernoff, é menor que $\delta^{1/4}$.

Por fim, o erro esperado para a estimativa do produto escalar entre vetores se dá pela desigualdade

$$A \cdot B \leq \widehat{A \cdot B} \leq A \cdot B + \epsilon\|A\|_1\|B\|_1$$

Para esta demonstração, definimos $(\widehat{A \cdot B})_q$ como a estimativa do produto escalar considerando apenas a linha q da matriz. Pela construção da matriz, pode-se dizer que

$$(\widehat{A \cdot B})_q = A \cdot B + \sum_{1 \leq i < j \leq n} I_{h_q,i,j} \cdot A[i] \cdot B[j]$$

Como ambos A e B possuem apenas elementos não-negativos, é fácil mostrar que tanto os valores de $(\widehat{A \cdot B})_q$ como a estimativa final (por consequência) são maiores que o valor

do produto escalar $A \cdot B$. Além disso,

$$\begin{aligned} E\left[\widehat{(A \cdot B)}_q - A \cdot B\right] &= E\left[\sum_{i,j} I_{h_q,i,j} \cdot A[i] \cdot B[j]\right] \\ &= \sum_{i,j} E\left[I_{h_q,i,j}\right] \cdot A[i] \cdot B[j] \\ &= \frac{\epsilon}{e}\|A\|_1\|B\|_1 \end{aligned}$$

Assim, a probabilidade de todas as funções *hash* produzirem estimativas acima do limite é dada por

$$\begin{aligned} \Pr\left[\widehat{A \cdot B} > A \cdot B + \epsilon\|A\|_1\|B\|_1\right] &= \Pr\left[\forall_q^k \widehat{(A \cdot B)}_q - A \cdot B > \epsilon\|A\|_1\|B\|_1\right] \\ &= \Pr\left[\forall_q^k \widehat{(A \cdot B)}_q - A \cdot B > eE\left[\widehat{(A \cdot B)}_q - A \cdot B\right]\right] \\ &= \left(\Pr\left[\widehat{(A \cdot B)}_q - A \cdot B > eE\left[\widehat{(A \cdot B)}_q - A \cdot B\right]\right]\right)^k \end{aligned}$$

logo, utilizando a desigualdade de Markov,

$$\Pr\left[\widehat{A \cdot B} > A \cdot B + \epsilon\|A\|_1\|B\|_1\right] < \frac{1}{e^k} \leq \delta$$

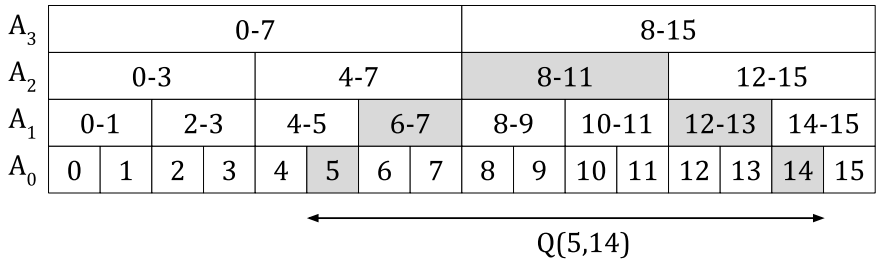
Perceba que este é um resultado compatível e mais geral à consulta de ponto no vetor. Neste caso, bastaria utilizar um vetor B, com apenas o índice i contendo o valor 1. Assim, o produto escalar deste com um vetor A arbitrário seria equivalente a obter $A[i]$.

2.2.4 Consultas de intervalo

É possível utilizar *Count-Min Sketch* para consultas de intervalo $Q(a, b) = \sum_{i=a}^{b} A[i]$ em um vetor $A[0..n-1]$. A ideia trivial seria fazer a estimativa de todos os valores $A[i]$ no intervalo selecionado. Porém, esta estratégia acumula erro proporcional ao tamanho do intervalo. Uma outra estratégia consiste manter não apenas um, mas $\log_2 n$ matrizes $M_y, 0 \leq y < \log_2 n$, representando implicitamente vetores A_y, onde $A_y[i] = \sum_{j=0}^{2^y-1} A[i+j]$, para todo i mútiplo de 2^y. Isto permite que qualquer consulta nos intervalos em $[0, n)$ seja efetuada com no máximo $2\log_2 n$ consultas nas matrizes subjacentes. A Figura 14 exemplifica uma dessas consultas. O valor $i - j$ em uma célula indica que seu conteúdo é $\sum_{i \leq k \leq j} A[k]$.

A atualização consiste em, ao receber uma tupla (i, c), atualizar cada uma das matrizes. Porém o índice utilizado em cada matriz é diferente. Seja i_y o índice usado para atualizar a matriz M_y, todos os bits de i_y devem ser iguais aos de i, exceto os y menos significativos, que devem ser iguais a zero. Por exemplo, para atualizar $i = 7$ num vetor com $n = 16$,

Figura 14: Consulta de intervalo $Q(5, 14)$ para $n = 16$



as seguinte posições devem ser atualizadas nos vetores implícitos: $i_0 = 7$, $i_1 = 6$, $i_2 = 4$ e $i_3 = 0$. O Algoritmo 12 mostra como se dá essa atualização.

Algoritmo 12 Atualiza Count-Min para busca por intervalo

```
1: procedimento ATUALIZAR(i, c)
2:     para y ← 0 to log2 n − 1 faça
3:         i_y ← i com os y bits menos significativos desligados
4:         para j ← 1 to k faça
5:             M_y[h_j(i_y), j] ← M_y[h_j(i_y), j] + c
6:         fim para
7:     fim para
8: fim procedimento
```

A consulta consiste decidir e estimar os valores para os sub-intervalos representáveis através dos vetores A_y que compõem o intervalo de busca. A decomposição em sub-intervalos pode ser feita de forma gulosa, em cada passo procurando o maior sub-intervalo que seja prefixo do intervalo atual. Para os fins do algoritmo, o mais importante é encontrar qual vetor A_y representa o intervalo em questão. O Algoritmo 13 mostra como funciona o processo de consulta.

Algoritmo 13 Estima o somatório dos elementos no intervalo

```
1: função ESTIMAR-INTERVALO(a, b)
2:     soma ← 0
3:     enquanto a ≤ b faça
4:         y ← maior índice de vetor A_y que representa algum prefixo de [a, b].
5:         minimo ← ∞
6:         para j ← 1 to k faça
7:             minimo ← min(minimo, M_y[h_j(a), j])
8:         fim para
9:         soma ← soma + minimo
10:        a ← a + 2^y
11:    fim enquanto
12:    retorna soma
13: fim função
```

Ainda seguindo o trabalho em [CM05], é possível mostrar que o erro desta estimativa

é dado por

$$Q(a,b) \leq \widehat{Q(a,b)} \leq Q(a,b) + 2\epsilon \log_2 n \|A\|_1$$

com probabilidade $1 - \delta$.

O princípio que rege este erro é o mesmo da consulta pontual. Neste caso, a ideia é mostrar que o erro de cada estimativa (somatório de até $2 \log_2 n$ observações independentes de $X_{h,i}$) tem valor esperado $2 \log_2 n(\epsilon/e)\|A\|_1$. Assim, aplicando a mesma desigualdade de Markov temos o resultado descrito acima.

É importante notar que na prática, o erro pode ser menor para intervalos cujos limites possuam um número menor de bits ligados, pois seriam necessários menos sub-intervalos para compor o intervalo da consulta.

2.2.5 Aplicações

A estrutura *Count-Min Sketch* permite a representação de vetores arbitrários sem o custo de mantê-los inteiramente em memória. Isto pemite a utilização de diversos algoritmos clássicos que seriam custosos em uma situação de restrição de recursos, mas que não sofreriam tanto em trocar acurácia das respostas por um uso mais controlado de memória.

Bioinformática: Em [ZPCK+14] é mostrado um caso onde a estrutura *Count-Min sketch* é usada para manter a contagem de *k-mers* (subsequências de K bases nitrogenadas em sequências de DNA). Estas frequências são usadas para alimentar outros algoritmos, que passam então a atuar de forma probabilística. O uso de estruturas probabilísticas neste caso é importante para aliviar o uso de memória para representar conjuntos de dados que não raramente chegam a dezenas de bilhões de registros.

Segurança: Muitos equipamentos de segurança possuem uma quantidade limitada de recursos computacionais. Isto torna atrativo o uso de estruturas de *sketch* para manter informações críticas em tempo real. Em [SVG08] é descrito um mecanismo que usa *Count-Min sketch* para verificar se uma certa combinação de parâmetros de conexão exibe uma frequência muito superior à média, o que pode ser um sinal de ataque.

Além disso, em [SHM], pesquisadores da Microsoft apresentam um mecanismo para manter uma estrutura com a frequência de senhas para invalidar senhas muito comuns, aumentando a segurança dos sistemas. A vantagem de usar *Count-Min* neste caso é capacidade de armazenar a frequência de uma senha sem manter a senha em si na estrutura.

Bancos de Dados: Um uso prático para a estrutura *Count-Min sketch* é a estimativa de joins relacionais em bancos distribuídos [CM05, RD07]. Neste uso, seria computada

a estrutura para o vetor de frequências dos diferentes valores que um certo atributo pode ter e, apenas transferindo o *sketch* é possível estimar a cardinalidade do join, através da estimativa do produto escalar entre os dois vetores.

Sistemas distribuídos: Encontrar certos elementos muito frequentes num fluxo de dados tem muitas aplicações práticas, como detecção de cenários de erros persistentes. Entretanto, algoritmos determinísticos para esse problema geralmente consistem em manter um contador para cada elemento distinto no conjunto. Utilizando a estrutura *Count-Min sketch* é possível derivar um algoritmo que estima com alta probabilidade quais são os elementos mais frequentes [ZOWX06]. Este algoritmo ainda tem a vantagem de poder ser executado de forma distribuída, pela própria natureza da estrutura de dados.

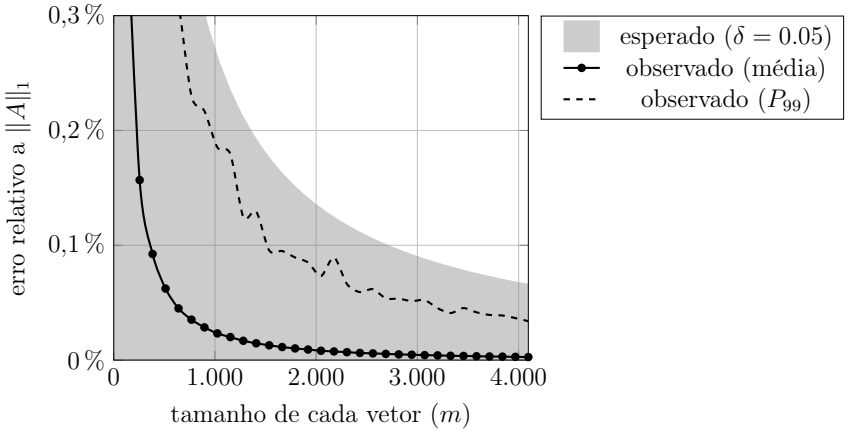
2.2.6 Resultados experimentais

Com o objetivo de testar as previsões teóricas sobre a estrutura *Count-Min sketch*, realizamos experimentos que verificam empiricamente os erros descritos anteriormente.

Cada teste usou uma variação do mesmo conjunto de dados composto por todas as obras de Shakespeare (42 obras, 964410 palavras, 23704 distintas). Em todos os casos, a família de funções *hash* utilizada foi MurmurHash 3 [App12], de 32 bits. Várias funções foram geradas, usando sementes diferentes. Para todos os testes, fixamos $k = 3$, o que resulta em uma confiança acima de 95% para todas as estimativas.

No primeiro teste, todas as palavras em todas as obras foram inseridas em *sketches* com m variando entre 128 e 4096. Depois, foram estimados as frequências de cada uma das palavras distintas bem como o erro relativo à norma $\|A\|_1$ do vetor. O resultado (média e percentil 99), bem como a previsão teórica, podem ser observado na Figura 15.

Figura 15: Erro observado por número de linhas da matriz para frequência

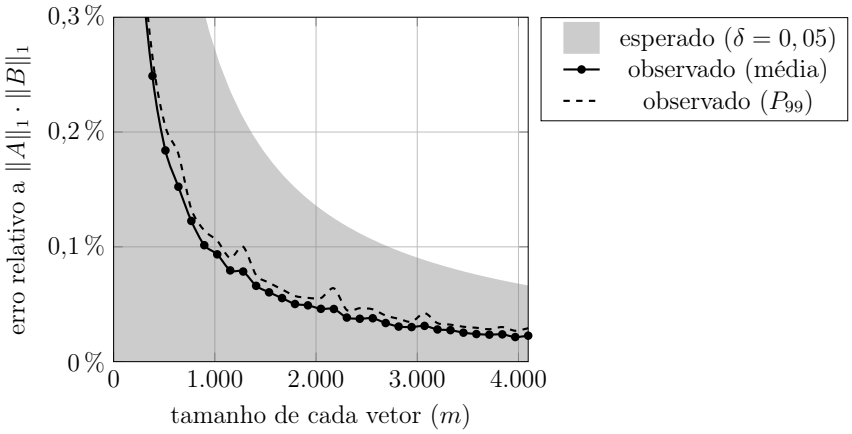


O segundo teste agrupou os textos por obra, em um total de 42 obras. Para cada obra, foi feita uma cópia deste conjunto com uma certa porcentagem aleatória de palavras

substituídas por strings aleatórias. 84 conjuntos de palavras foram utilizados no total.

Para cada conjunto de palavras foram computados *Count-Min sketches* do vetor de frequências com m variando entre 128 e 4096. Para cada par de conjuntos, foi estimado o produto escalar e seu erro (relativo a $\|A\|_1 \cdot \|B\|_1$). O resultado (média e percentil 99) pode ser observado na Figura 16.

Figura 16: Erro observado por número de linhas da matriz para produto escalar



Estes resultados mostram que os limites teóricos calculados são relativamente conservadores, o que pode ser explicado pelo uso da desigualdade de Markov, que impõe um limite fraco sobre a probabilidade de dispersão.

2.3 *MinHash*

MinHash é um algoritmo *hashing* sensível a localização que permite estimar a semelhança entre conjuntos de forma indireta, através da aproximação do coeficiente de similaridade de *Jaccard*.

O algoritmo foi inventado por Andrei Broder [Bro97] como forma de detectar páginas quase-duplicadas no mecanismo de busca Alta Vista. Antes disso, Heintze [H$^+$96] e Manber [M$^+$94] já haviam descrito um mecanismo de *fingerprinting* de documentos para rápida indexação e busca por similaridade. Entretanto, somente com o artigo de Broder, a técnica ganhou mais notoriedade.

MinHash possui muitas aplicações. A principal envolve a detecção de documentos quase duplicados, através da representação dos mesmos como o conjunto de palavras ou n-gramas que contêm. O algoritmo, entretanto, possui aplicações diversas, desde sistemas de recomendação [DDGR07] até técnicas processamento de som [CBWW10, CB07].

2.3.1 Definição

O objetivo do algoritmo *MinHash* é estimar o coeficiente de Jaccard. Este coeficiente é definido, para dois conjuntos A e B, como a razão $J(A, B)$ entre a cardinalidade da

interseção e a cardinalidade da união dos conjuntos [RV96]. Isto é,

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

O coeficiente $J(A, B)$ assume valores entre 0 e 1, sendo 0 para A e B disjuntos e 1 para A e B idênticos. Por exemplo, para os conjuntos:

$$A = \{1, 3, 7, 14, 20\} \text{ e}$$
$$B = \{1, 3, 7, 19, 20, 35\},$$

o valor do coeficiente é:

$$J(A, B) = \frac{|\{1, 3, 7, 20\}|}{|\{1, 3, 7, 14, 19, 20, 35\}|} = \frac{4}{7}$$

Embora seja trivial calcular o coeficiente de *Jaccard*, pode ser computacionalmente custoso realizar a comparação entre muitos pares de conjuntos com números muito grandes de elementos. Por isso, pode ser vantajoso pré-processar informações para cada conjunto que auxiliem na posterior aproximação do coeficiente. Em sua variante básica, tal pré-processamento consiste em aplicar um número constante de funções *hash* sobre cada elemento de um conjunto e guardar o mínimo valor obtido para cada função. O resultado obtido é chamado de *assinatura* do conjunto. Usando as assinaturas de cada conjunto é possível estimar a semelhança entre eles em tempo constante [Bro97], como veremos logo adiante.

A matriz característica de conjuntos $S_1, \ldots, S_n$ é uma matriz binária M na qual cada linha está mapeada a um elemento de $S_1 \cup S_2 \cup \cdots \cup S_n$ e a i-ésima coluna está mapeada a S_i. Cada posição na matriz é tal que $M[x, i] = 1$ se $x \in S_i$, para todo $x \in S_1 \cup S_2 \cup \cdots \cup S_n$. Considere, por exemplo, os conjuntos

$$S_1 = \{a, d\},\ S_2 = \{c\},\ S_3 = \{b, d, e\} \text{ e } S_4 = \{a, c, d\}.$$

Portanto, $S_1 \cup S_2 \cup S_3 \cup S_4 = \{a, b, c, d, e\}$ e uma matriz característica associada é aquela da Tabela 3.

Tabela 3: Matriz característica para os conjuntos S_1, S_2, S_3 e S_4.

elemento	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

Esta matriz característica geralmente não é a estrutura de dados usada na implemen-

tação dos conjuntos, e sim apenas uma forma conveniente de representá-los para facilitar a compreensão do algoritmo.

Para computar a assinatura de um conjunto, primeiro obtemos uma matriz característica com uma permutação aleatória de linhas. Assim, a assinatura $h_{\min}(S)$ de um certo conjunto S é definida pelo primeiro elemento na permutação que pertence a S.

Considere no exemplo a permutação (b, e, a, d, c). A partir dela, podemos definir a matriz característica mostrada na Tabela 4.

Tabela 4: Matriz permutada, destacando o h_{min} de cada conjunto.

elemento	S_1	S_2	S_3	S_4
b	0	0	**1**	0
e	0	0	1	0
a	**1**	0	0	**1**
d	1	0	1	1
c	0	**1**	0	1
h_{min}	a	c	b	a

Broder [Bro97] mostra que, para dois conjuntos A e B, e um *min hash* $h_{\min}$ aleatoriamente escolhidos, a probabilidade de terem o mesmo valor para $h_{\min}$ é igual ao próprio índice de *Jaccard*, conforme se demonstra a seguir.

Parte-se do princípio de que, considerando as colunas para os conjuntos A e B na matriz característica, os conjuntos X, Y e Z particionam o conjunto das linhas de M: onde ambas as colunas têm valor 1 (subconjunto X de linhas), onde cada coluna tem um valor diferente (subconjunto Y de linhas) e onde ambas as linhas têm valor 0 (subconjunto Z de linhas).

Por um lado, $J(A, B) = |X|/(|X| + |Y|)$, pois X representa $A \cap B$ e Y representa $A \cup B - A \cap B$. Por outro lado, dada uma permutação aleatória da matriz, a probabilidade de uma linha em X (i.e. $h_{\min}(A) = h_{\min}(B)$) aparecer antes de uma linha do tipo Y (i.e. $h_{\min}(A) \neq h_{\min}(B)$) é exatamente $|X|/(|X| + |Y|)$. Portanto,

$$\Pr[h_{\min}(A) = h_{\min}(B)] = J(A, B)$$

É possível, assim, definir um estimador não-enviesado para o índice de *Jaccard*

$$\hat{J}(A, B) = \mathbb{1}(h_{\min}(A) = h_{\min}(B))$$

onde

$$\mathbb{1}(b) = \begin{cases} 1 & \text{se } b = \textsc{Verdadeiro} \\ 0 & \text{se } b = \textsc{Falso} \end{cases}$$

Este estimador, entretanto, assume apenas os valores 0 ou 1. Possui, portanto, uma grande variância em relação ao valor esperado. Entretanto, serve como ponto de partida

para as duas principais variantes do algoritmo, como veremos a seguir.

2.3.2 Variante com múltiplas funções *hash*

Uma variante simples do algoritmo *MinHash* utiliza múltiplas funções *hash* para gerar vários estimadores e, através da média simples entre eles, estimar o índice de *Jaccard*.

Isto é, utilizam-se k funções *hash* $\{h_1, h_2, ..., h_k\}$. Cada função mapeia elementos do conjunto no intervalo [0..1]. Com isso, define-se o estimador:

$$\hat{J}(A,B) = \frac{1}{k}\sum_{i=1}^{k} \mathbb{1}(h_{i,\min}(A) = h_{i,\min}(B))$$

O algoritmo em si consiste em computar uma assinatura H utilizando k funções *hash* para cada conjunto. Esta assinatura será comparada, valor a valor, para estimar a semelhança entre os conjuntos.

Algoritmo 14 Computa a assinatura de um conjunto S

```
1: função COMPUTAR-ASSINATURA(S)
2:     para i ← 1 to k faça
3:         H[i] ← ∞
4:         para each x ∈ S faça
5:             H[i] ← min(H[i], h_i(x))
6:         fim para
7:     fim para
8:     retorna H
9: fim função
```

Uma vez computada a assinatura, para comparar dois conjuntos basta verificar quantos *min hash* são comuns entre eles. O resultado final, assumindo que y elementos são comuns, se dá por y/k (Algoritmo 15).

Algoritmo 15 Compara assinaturas de conjuntos

```
1: função COMPARAR-ASSINATURAS(H_1, H_2)
2:     y ← 0
3:     para i ← 1 to k faça
4:         se H_1[i] = H_2[i] então
5:             y ← y + 1
6:         fim se
7:     fim para
8:     retorna y/k
9: fim função
```

A complexidade de tempo de cada parte do algoritmo é simples de determinar. Ao computar a assinatura, para cada elemento do conjunto, k funções *hash* são calculadas.

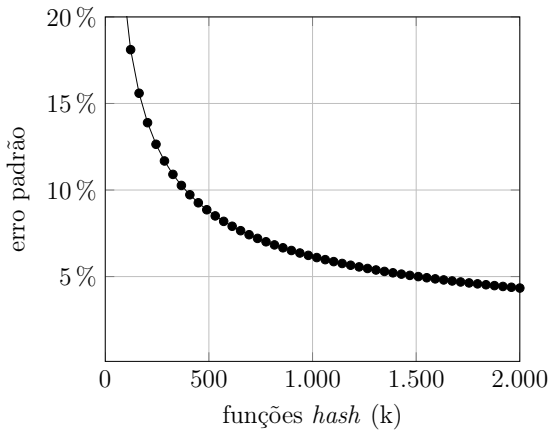
Assim, para um conjunto com n elementos, a complexidade é $O(nk)$. Para comparar duas assinaturas, apenas $O(k)$ operações são feitas.

Para calcular o erro provável, cada estimador pode ser visto como uma variável aleatória de Bernoulli com $J(A, B)$ probabilidade de ser 1. Assim, o erro do estimador composto pode ser facilmente calculado aplicando o limite de Chernoff [CDF+01, TSMJ12], que determina que, para haver erro menor que θ, com confiança de $1 - \delta$, é preciso escolher k tal que

$$k \geq \frac{2 + \theta}{\theta^2} \times \ln(2/\delta)$$

Por exemplo, a Figura 17 mostra o gráfico de erro padrão por funções *hash*.

Figura 17: Erro padrão por funções *hash*



É razoável argumentar que para representar conjuntos arbitrários de cardinalidade n, seria necessário usar funções *hash* de $O(\log n)$ bits. Entretanto, em [LK10], Li e König argumentam que para qualquer esquema de *hash* sensível a localidade (como *MinHash*), guardar apenas um número constante dos bits menos significativos dos *hashes* calculados não aumenta consideravelmente a variância dos estimadores para valores de similaridade próximos a 0,5.

2.3.3 Variante com apenas uma função *hash*

Muitas vezes, o custo de computar várias funções *hash* pode ser muito alto na prática, especialmente para conjuntos com centenas de milhões de elementos.

Uma variante possível do *MinHash* é utilizar apenas uma função *hash* e manter os k menores resultados em ordem ascendente de $h(x)$ para cada conjunto como assinatura (Algoritmo 16). Denota-se por $h_{(k)}(S)$ o conjunto com k menores hashes do conjunto S e $H_{\max}$ o valor máximo em H.

A similaridade entre os conjuntos será computada de uma forma diferente da variante com múltiplas funções *hash*. Nesta, vale-se do princípio de que os k menores elementos em $h_{(k)}(A) \cup h_{(k)}(B)$ são os mesmos que em $h_{(k)}(A \cup B)$. Assim, seja $Y = h_{(k)}(A \cup$

Algoritmo 16 Computa a assinatura de um conjunto S

```
1: função COMPUTAR-ASSINATURA(S)
2:     H ← ∅
3:     para each e ∈ S faça
4:         H ← H ∪ h(e)
5:         se |H| > k então
6:             H ← H − {H_max}
7:         fim se
8:     fim para
9:     retorna H
10: fim função
```

$B) \cap h_{(k)}(A) \cap h_{(k)}(B)$. Y equivale aos membros de $h_{(k)}(A \cup B)$ que também estão em $h_{(k)}(A \cap B)$. Pode-se, então, definir $|Y|/k$ como um estimador não-enviesado de $J(A, B)$ (Algoritmo 17).

Algoritmo 17 Estima $J(A.B)$, sendo H_1 e H_2 as assinaturas, respectivamente, de A e B

```
1: função COMPARAR-ASSINATURAS(H_1, H_2)
2:     H_x ← k menores elementos de H_1 ∪ H_2
3:     H_y ← H_x ∩ H_1 ∩ H_2
4:     retorna |H_y|/k
5: fim função
```

Apesar da diferença prática, a ideia é similar à da variante com múltiplas funções *hash*. Entretanto, nesta variante é preciso considerar que, em vez de obter apenas o primeiro elemento de cada permutação da matriz característica, obtém-se k elementos. Mesmo assim, também é possível usar o limite de Chernoff para estimar o erro, pois o mesmo resultado para amostragem com substituição pode ser usado para amostragem sem substituição, como mostra Hoeffdin [Hoe63, BM$^+$15].

2.3.4 Exemplo

Com o objetivo de facilitar o entendimento da estrutura *MinHash*, apresentamos aqui um exemplo prático de seu uso. Consideraremos aqui o cálculo da assinatura *MinHash* de dois conjuntos A e B, ambos contendo seis strings. Tal assinatura será computada através da variante com múltiplas funções *hash*. São definidas $k = 8$ funções *hash* de 8 bits (por simplicidade). A Figura 18 mostra as tabelas que dão origem às assinaturas dos conjuntos.

Cada linha de cada tabela corresponde a uma string, bem como os resultados da aplicação de cada função $h_1, \ldots, h_8$, no intervalo $[0; 255]$ (pois a função escolhida retorna valores de 8 bits). O rodapé da tabela mostra os valores mínimos para cada função. Esses valores compõem a assinatura de cada conjunto. A estimativa do índice de Jaccard se dá obtendo a proporção das posições nas assinaturas dos dois conjuntos que possuem o

Figura 18: Exemplos de construção de assinatura *MinHash*

A	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8
Arthur Dent	197	98	11	86	71	82	57	204
Tricia McMillan	137	68	172	6	132	40	238	106
Zaphod Beeblebrox	247	6	253	138	225	220	229	120
Ford Prefect	71	182	221	96	237	63	209	201
Marvin	11	193	29	192	186	182	113	150
Fenchurch	118	207	1	251	80	34	116	143
MinHash(A)	11	6	1	6	71	34	57	106

B	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8
Zaphod Beeblebrox	247	6	253	138	225	220	229	120
Ford Prefect	71	182	221	96	237	63	209	201
Marvin	11	193	29	192	186	182	113	150
Fenchurch	118	207	1	251	80	34	116	143
Slartibartfast	208	8	198	81	195	181	195	153
Prostetnic Vogon Jeltz	171	178	130	196	145	237	73	88
MinHash(B)	11	6	1	81	80	34	73	88

mesmo valor. No exemplo, metade dos valores são iguais entre as assinaturas (respectivos às funções h_1, h_2, h_3 e h_6). Considerando que no exemplo, de fato, $J(A, B) = 4/8 = 0,5$, podemos dizer que a estimativa foi precisa.

2.3.5 Detecção de quase-duplicatas

A motivação inicial que levou à criação do algoritmo *MinHash* era encontrar quase-duplicatas em uma coleção de 30 milhões de documentos indexados pelo motor de busca Alta Vista em 1997 [Bro97]. Percebe-se que apenas considerando a técnica *hashing*, a solução ainda é bastante impraticável, pois uma busca par-a-par no conjunto de dados requer $\binom{30,000,000}{2}$ – ou aprox. 450 trilhões – operações. Mesmo sendo capaz de executar cada comparação em 1 microsegundo, ainda levaria mais de uma década para processar a coleção inteira.

É importante notar, entretanto, que se o objetivo é encontrar grupos de similaridade, não é necessário computar o índice para todos os pares de conjuntos. É suficiente focar nos pares que possuem maior probabilidade de serem similares.

É possível utilizar a teoria *hashes* sensíveis a localidade para diminuir a quantidade de pares a serem verificados. Uma técnica aplicável neste cenário é dividir as funções *hash* em bandas e separar as assinaturas em baldes baseados no valor da assinatura em cada banda isoladamente. Assinaturas similares terão uma tendência maior de serem colocadas no mesmo balde (por terem valores idênticos de assinatura naquela banda), com uma probabilidade definida.

Para entender a técnica, considere múltiplos conjuntos e a matriz formada por suas assinaturas *MinHash*, utilizando a variante com múltiplas funções *hash*. Cada coluna representa um conjunto e cada linha representa uma função *hash*. O objetivo é separar as linhas em bandas e, para cada banda, verificar os grupos de assinaturas formados como candidatos a duplicatas.

Por exemplo, na Tabela 5 ilustra-se uma matriz de assinatura representando 5 conjuntos e 8 funções *hash*, divididas em 4 bandas com 2 linhas cada.

No exemplo, a banda 2 revela um potencial par de quase-duplicatas, pois S_2 e S_5

Tabela 5: Matriz de assinaturas

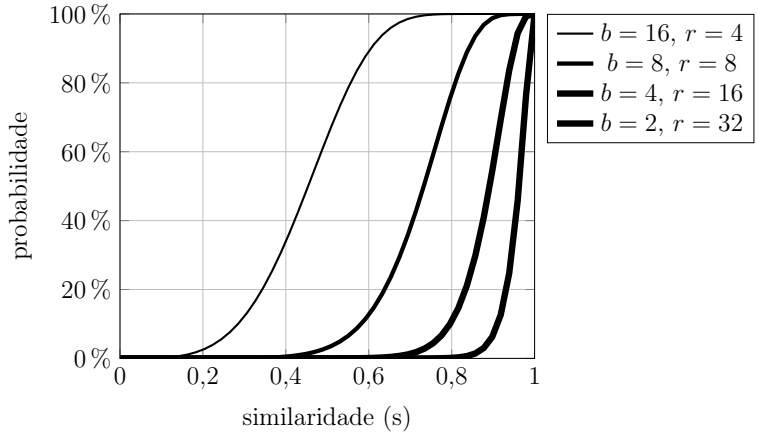
banda	hash	S_1	S_2	S_3	S_4	S_5
1	h_1	6	1	7	6	2
	h_2	1	3	7	1	3
2	h_3	8	3	8	5	3
	h_4	0	9	4	1	9
3	h_5	2	0	6	2	0
	h_6	0	0	3	1	0
4	h_7	5	1	1	5	1
	h_8	4	4	9	4	4

possuem o mesmo valor para suas assinaturas naquela banda.

Considerando que deseja-se encontrar pares com similaridade acima de um certo limite, esta técnica está sujeita tanto a falsos positivos quantos falsos negativos. A probabilidade de um par de conjuntos A e B, com $J(A, B) = s$, ser marcado como potencial duplicata, numa matriz com b bandas com r linhas por banda é exatamente a probabilidade dos dois conjuntos concordarem em todas as linhas de pelo menos uma banda, que é de $1-(1-s^r)^b$ [RUUU12].

Baseado nesta probabilidade, a Figura 19 mostra, em função da similaridade, a probabilidade de um par de documentos ser marcado como duplicado em uma matriz com 64 funções *hash* e algumas escolhas de b diferentes.

Figura 19: Probabilidade de ser escolhido como duplicata



Independente dos valores de b e r escolhidos, o gráfico sempre terá esta forma de "S". Entretanto, estes parâmetros definem com qual probabilidade um par de certa similaridade s é considerado duplicado. Ainda assim, valores abaixo da similaridade definida podem ser considerados duplicados (falsos positivos) e valores acima podem ser ignorados (falsos negativos).

A partir de b e r e uma probabilidade p é possível derivar a partir de qual valor de similaridade se possui aquela probabilidade de ser escolhido, por inversão da função que

fornece esta probabilidade:

$$s = \left(1 - (1-p)^{1/b}\right)^{1/r}$$

Por exemplo, $b = 32$, $r = 16$, temos 0,5% de chance de encontrar um par de conjuntos com similaridade acima de $0,5683$, e 95% de chance de encontrar pares com similaridade acima de $0,8891$. Ao manipular estes valores é possível dimensionar facilmente a tolerância a falsos positivos ou falsos negativos no algoritmo.

2.3.6 *SimHash*

A partir do trabalho de Indyk e Motwani [IM98, GIM+99], começou-se a formalizar as bases teóricas das técnicas *hashing* sensível a localidade. De fato, *MinHash* é apenas um tipo destes hashes. Muitos outros tipos foram estudados desde então. Uma dos mais bem sucedidos é o *SimHash* [Cha02]. Este algoritmo ganhou notoriedade nos últimos anos por compor um dos fatores usados pelo Google para priorizar a indexação de páginas na Internet [MJDS07].

Um hash sensível a localidade é definido por uma família de funções *hash* $\mathcal{F}$, tal que para dois elementos $x, y \in U$, e uma certa função de similaridade $sim : U \times U \to [0,1]$, vale

$$\Pr_{h \in \mathcal{F}}[h(x) = h(y)] = sim(x,y)$$

Em especial, para o *MinHash*, o objetivo é estimar a similaridade entre conjuntos com um número grande de elementos, aproximando o índice de Jaccard, i.e. $sim(x,y) = J(x,y)$. Outras medidas podem ser utilizadas.

No caso das assinaturas de Charikar – ou *SimHash* – o objetivo é estimar a similaridade entre vetores em espaços de alta dimensão. Neste caso utiliza-se uma família de funções *hash* baseadas no produto escalar entre vetores. No algoritmo, para comparar a similaridade em vetores em $\mathbb{R}^d$, escolhe-se um vetor $\vec{r}$, de d dimensões, onde cada coordenada é um valor escolhido aleatoriamente em uma distribuição gaussiana. Define-se a função como

$$h_{\vec{r}}(\vec{u}) = \begin{cases} 1 & \text{se } \vec{r} \cdot \vec{u} \geq 0 \\ 0 & \text{se } \vec{r} \cdot \vec{u} < 0 \end{cases}$$

Goemans e Williamson [GW95] mostram que esta função pode ser utilizada como um hash sensível a localidade, tal que, para vetores $\vec{u}$ e $\vec{v}$,

$$\Pr[h_{\vec{r}}(\vec{u}) = h_{\vec{r}}(\vec{v})] = 1 - \frac{\theta(\vec{u},\vec{v})}{\pi},$$

onde $\theta(\vec{u},\vec{v})$ é o menor ângulo formado por $\vec{u}$ e $\vec{v}$.

É possível utilizar esta família de funções para estimar a similaridade entre conjuntos. Cada elemento da união dos dois conjuntos seria associado a uma dimensão e cada con-

junto representado como um vetor, tendo valor 1 nas dimensões respectivas aos elementos que contém. Por exemplo, sejam dois conjuntos A e B:

$$A = \{a, d, e\} \text{ e } B = \{b, c, d, e\}$$

uma possível definição dos vetores relativos a A e B seria:

$$v_A = (1, 0, 0, 1, 1) \text{ e } v_B = (0, 1, 1, 1, 1)$$

Como o menor ângulo formado por estes dois vetores é 1,15026 rad, então a similaridade entre os dois, segundo esta métrica, é igual a aproximadamente 0.695913. Para conjuntos, esta função *hash* (tal associação de vetores a elementos) representa a seguinte métrica de similaridade:

$$\Pr[h_{\vec{r}}(\vec{u}_A) = h_{\vec{r}}(\vec{u}_B)] = 1 - \frac{\arccos\left(\frac{|A \cap B|}{\sqrt{|A| \cdot |B|}}\right)}{\pi}.$$

Na prática, o algoritmo consiste em computar k bits, armazenados em um vector $V[1..k]$, onde o elemento $V[i]$ assume valor 0 se, usando-se a i-ésima função *hash*, houver menos elementos no conjunto com valor *hash* negativo do que não-negativo, ou assume valor 1 caso contrário (Algoritmo 18).

Algoritmo 18 Computa a assinatura *SimHash* de um conjunto S

```
1: função COMPUTAR-ASSINATURA(S)
2:     para i ← 1 to k faça
3:         v ← 0
4:         para each e ∈ S faça
5:             se h_i(e) ≥ 0 então
6:                 v ← v + 1
7:             senão
8:                 v ← v − 1
9:             fim se
10:        fim para
11:        H[i] ← (v ≥ 0)
12:    fim para
13:    retorna H
14: fim função
```

A comparação entre duas assinaturas consiste em determinar a proporção de bits iguais nas assinaturas (Algoritmo 19).

Henzinger [Hen06] argumenta que *SimHash* tem um desempenho melhor ao estimar quase-duplicatas em documentos na Internet, se comparado ao *MinHash*. Em especial, *SimHash* requer menos espaço para atingir mesma precisão que *MinHash*. Por outro

Algoritmo 19 Compara assinaturas *SimHash* de conjuntos

```
1: função COMPARAR-ASSINATURAS(H_1, H_2)
2:     y ← 0
3:     para i ← 1 to k faça
4:         se H_1[i] = H_2[i] então
5:             y ← y + 1
6:         fim se
7:     fim para
8:     retorna y/k
9: fim função
```

lado, Shrivastava e Li [SL14] afirmam que *MinHash* é mais adequado que *SimHash* para coleções de documentos com muitas similaridades.

Manku et al. [MJDS07] descrevem a aplicabilidade desta técnica para detecção de quase-duplicatas usando assinaturas de 64 bits num banco de dados de 8 bilhões de páginas. Também sugerem um algoritmo otimizado para encontrar todas as assinaturas que diferem de uma assinatura específica em no máximo k bits, onde k é um inteiro pequeno.

2.3.7 Aplicações

O algoritmo *MinHash* e outros mecanismos *hash* sensíveis a localidade, têm enorme aplicação prática, especialmente como forma de oferecer uma função de similaridade e algoritmos de clusterização para os mais variados fins. Nesta seção listaremos alguns exemplos de aplicações comuns nesta área.

Sites de busca: Em seu trabalho seminal sobre *MinHash*, Broder [Bro97] já citava uma aplicação prática na indexação de páginas web pelo motor do Alta Vista, numa análise investigativa a fim de determinar quase-duplicatas em um índice de 30 milhões de documentos.

Manku et al. [MJDS07] descrevem como usam, no Google, outra variante *hash* sensível a localidade chamada *SimHash*, baseada em distância entre vetores. No artigo, os autores introduzem uma técnica para determinar, a partir de coleção de 8 bilhões de páginas com assinaturas *SimHash* de 64 bits pré-calculadas, se uma nova página encontrada pelo indexador possui uma duplicata já indexada.

Bancos de dados de imagens: Embora hashing sensível a localidade seja mais comumente usado para comparação de documentos textuais, também é possível adaptá-lo para comparar a semelhança entre imagens. Neste caso, várias caracteristicas da imagem podem ser usadas como meio de comparação, desde histogramas de cores, parâmetros de iluminação, até os pixels individuais.

Ioffe [Iof10] cita um caso de uso para uma variante do algoritmo *MinHash*, modificada para permitir conjuntos ponderados, de modo a aproximar a distância ℓ_1

entre vetores. No artigo, o objetivo principal é apresentar o uso da técnica, no Google, para busca aproximada por imagens num banco de dados. As imagens são representadas por vetores de features, como histogramas de cores e metadados.

Lee et al. [LKI10] também descrevem um método para busca parcial de imagens usando *MinHash*. Neste caso, o objetivo é agrupar imagens que provavelmente contém um mesmo objeto, mesmo que as imagens não sejam inteiramente compostas por ele.

Numa aplicação mais clássica, Wang et al. [WZL13] expõem os resultados alcançados pela divisão Microsoft Research em uma busca por duplicatas em uma coleção de mais de 2 bilhões de imagens da Internet. Utilizando uma variante em dois passos do *MinHash*, eles foram capazes de encontrar mais de 500 milhões de imagens duplicadas em 13 horas de processamento em um cluster de 2.000 núcleos.

Sistemas de recomendação: A capacidade de verificar a semelhança entre conjuntos ou vetores abre portas para aplicação de *MinHash* em algoritmos de clusterização baseados em similaridade.

O serviço de notícias Google News utiliza *MinHash* para seu mecanismo de recomendação de artigos para os usuários [DDGR07]. A recomendação é calculada em tempo real com latência abaixo de um segundo. Para tanto, os autores descrevem como utilizam aprendizado de máquina, aplicando *MinHash* como função de similaridade. Todo o processamento é realizado no modelo MapReduce, para permitir a criação de um mecanismo de recomendação de notícias com alta escalabilidade.

Além disso, Rodrigues [R$^+$13] sugere a utilização de *MinHash* para clusterização de espectadores em serviços de TV, baseados nas opções pregressas.

Similaridade em Redes Sociais: A detecção de comunidades orgânicas em redes sociais é um problema cada vez mais comum atualmente. O problema pode ser modelado como um caso especial de sistema de recomendação, onde o objetivo é clusterizar nós comuns num grafo por um conjunto de características.

Macropol e Singh [MS10] introduzem em 2010 o algoritmo *Top Graph Clusters* (*TopGC*), utilizando hashes sensíveis a localidade – especialmente *MinHash* – para detecção, em tempo linear, de subgrafos altamente conectados. O algoritmo propõe utilizar, como métrica de afinidade entre os nós, a semelhança entre suas vizinhanças.

Teixeira et al. [TSMJ12] descrevem como utilizar *MinHash* como função de similaridade, com o objetivo de computar a semelhança entre grafos utilizando poucos recursos.

2.3.8 Resultados experimentais

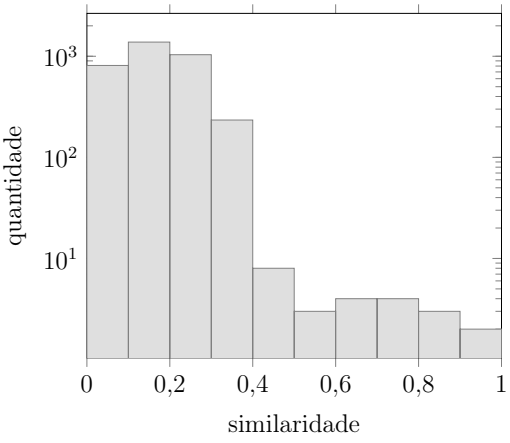
Para melhor observar as previsões teóricas sobre o algoritmo *MinHash*, conduzimos uma série de experimentos para verificar empiricamente as probabilidades descritas na teoria. Testamos tanto a variante com múltiplas funções *hash* quanto com apenas uma função. Testamos também a técnica de clusterização de pares duplicados descrita na Seção 2.3.5.

Para todos os testes, foi utilizado um conjunto de dados misto, composto por todas as obras de Shakespeare (42 obras, 964410 palavras, 23704 distintas). Para cada obra, consta no conjunto de dados o conjunto de palavras distintas da obra, bem como uma cópia deste conjunto com uma certa porcentagem aleatória de palavras substituídas por strings aleatórias (84 conjuntos de palavras foram utilizados no total).

Em todos os casos, a família de funções *hash* utilizada foi MurmurHash 3 [App12], de 32 bits. Várias funções foram geradas, usando sementes diferentes.

A métrica de similaridade utilizada no teste foi o índice de Jaccard entre os conjuntos simples de palavras de cada texto. Este índice foi calculado de forma determinística para os $\binom{42\times 2}{2} = 3486$ pares de documentos. A distribuição de similaridades entre os pares pode ser vista no histograma da Figura 20.

Figura 20: Distribuição de similaridades entre pares de documentos



Para os dois testes, variou-se k (o número de funções *hash*) entre 25 e 1000. Mediu-se então, para cada par, o quanto o estimador da similaridade desviava do valor real (calculado deterministicamente). Para estes valores foram computados a média, e o desvio padrão. A Figura 21 mostra os resultados obtidos para as variantes de múltiplas funções e apenas uma função *hash*, respectivamente. Na figura é possível comparar com o intervalo esperado pela teoria apresentada, com 95% de certeza.

Também foi testada a técnica de detecção de duplicatas descrita na Seção 2.3.5.

Inicialmente calculou-se, para cada documento, assinaturas *MinHash* com 512 funções *hash*. Utilizou-se então a técnica de detecção de duplicatas utilizando todas as possíveis combinações de número de bandas (b) e linhas por banda (r).

Figura 21: Erro observado por funções *hash*

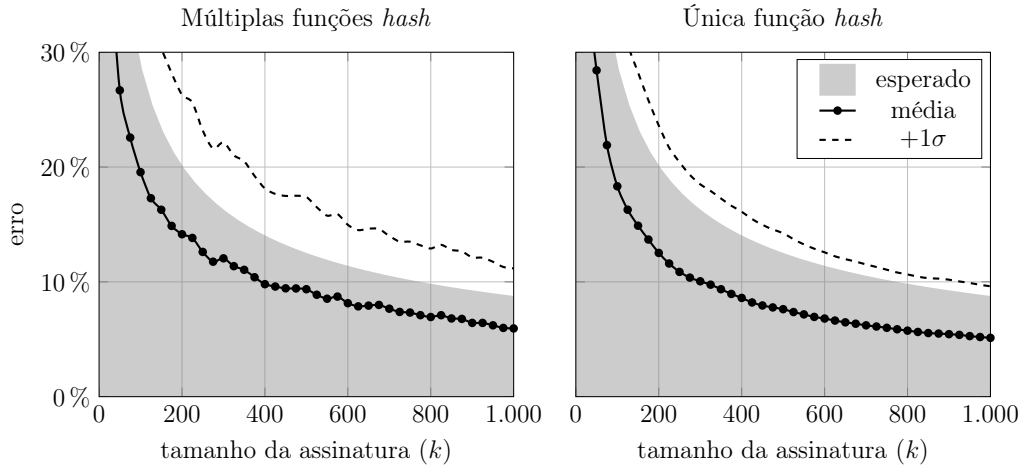
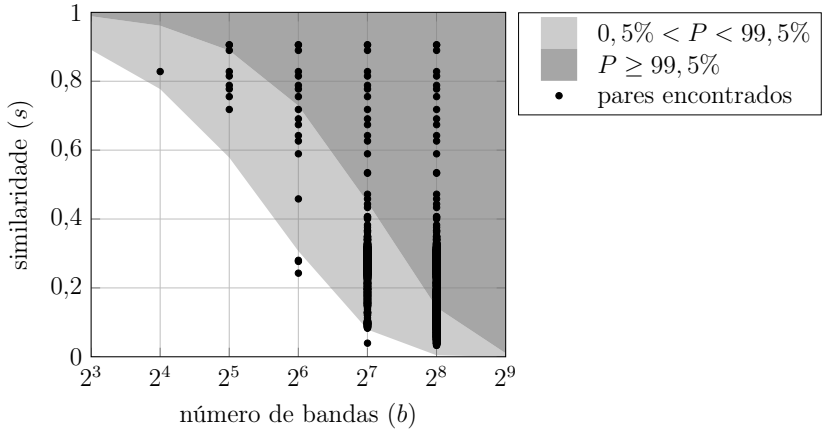


Figura 22: Pares detectados para cada configuração de bandas.



As similaridades dos pares encontrados com cada configuração, bem como a comparação com o valor predito pela teoria podem ser vistos na Figura 22. Configurações que resultaram em nenhum par encontrado são omitidas por brevidade.

É importante notar no resultado não só valores abaixo da probabilidade de corte (falsos positivos), como muitos valores de similaridade altos omitidos numa certa configuração de bandas, mas que aparecem na próxima (falsos negativos).

2.4 *HyperLogLog*

Consideremos o problema de determinar o número de elementos distintos em um multiconjunto. Na prática, os elementos destes multiconjuntos podem ser identificadores de usuários, endereços IP, pacotes de rede, etc. Geralmente o desejável é encontrar a cardinalidade efetuando apenas uma passagem pelos dados, utilizando o mínimo de memória possível [MAA08, CC12]. Na Seção 2.1.4, discutimos a aplicabilidade dos filtros de Bloom para estimativa de cardinalidade em fluxos de dados. Existem, entretanto, outros algoritmos mais eficientes para este problema, que é recorrente na análise de grandes massas de

dados. *HyperLogLog* é um algoritmo que permite estimar o número de elementos distintos em um fluxo de dados, utilizando memória sublinear. É possível estimar a cardinalidade de elementos distintos em um conjunto com bilhões de elementos, com 2% de erro padrão, utilizando apenas 1,5KB de memória.

Um algoritmo determinístico precisaria de memória proporcional ao número de elementos, pois todos precisariam ser armazenados. Entretanto, para muitas aplicações é possível relaxar os requisitos de precisão de forma a permitir algoritmos que utilizam menos recursos.

Apesar de *Linear Counting* [WVZT90] já ser conhecido desde o início dos anos 90, sua complexidade de memória ainda é linear em relação ao número de elementos do conjunto. Para aplicações com grandes volumes de dados em tempo real, uma solução sublinear se mostra mais apropriada.

Flajolet e Martin [FM85], em seu trabalho seminal na década de 80, descrevem um algoritmo para estimativa de cardinalidade que se baseia na observação do padrão de bits do hash dos elementos. Este trabalho é conhecido como *Probabilistic Counting* e foi fortemente inspirado pela ideia de Morris [Mor78] alguns anos antes.

Em 1996, Alon, Matias e Szegedy [AMS96] consolidam a teoria para complexidade de tempo e espaço para estimativa de *momentos de frequência*, que são definidos a seguir. Para um conjunto $A = \{a_1, a_2, \ldots, a_n\}$, onde cada a_i corresponden à frequência de um elemento distinto de um multiconjunto S com n elementos distintos, o momento de frequência $F_k(S)$ é definido como:

$$F_k(S) = \sum_{i=1}^{n} a_i^k$$

Perceba que F_0 corresponde à cardinalidade de elementos distintos do multiconjunto, assim como F_1 corresponde à soma destes elementos. Em [AMS96], os autores concluem que F_0, F_1 e F_2 podem ser aproximados com complexidade de espaço logarítmica. A partir deste resultado, diversos algoritmos foram desenvolvidos com o objetivo de estimar cardinalidades em multiconjuntos. Os algoritmos se dividem em duas grandes categorias, dependendo se são baseados em estatísticas de ordem ou padrão de bits. [FFGM08, CC12].

Algoritmos baseados em estatísticas de ordem

Baseiam-se na probabilidade de um certo hash ter uma posição específica na ordem definida pela função *hash* escolhida. Por exemplo, se o menor hash entre todos em um conjunto, considerando uma distribuição uniforme no intervalo $[0; 1]$, for igual a $0,05$, espera-se que a cardinalidade do conjunto seja da ordem de 20. Algoritmos como *K-Minimum Values* [BYJK$^+$02] e *MinCount* [Gir09] baseiam-se neste princípio.

Algoritmos baseados em padrões de bits

Baseiam-se na probabilidade de certos padrões de bits acontecerem no hash de elementos do conjunto. Por exemplo, observando os hashes de uma sequência, se for visto um valor cuja representação binária comece com $0^{p-1}1$, é provável que a cardinalidade seja da ordem de 2^p. Algoritmos como o já citado *Probabilistic Counting* [FM85], *LogLog* [DF03] e *HyperLogLog* [FFGM08] são desta categoria.

Se apenas um estimador for mantido, a variância será muito grande para ser útil. Por isso é importante manter múltiplos estimadores. Se o erro padrão de um estimador for igual a σ, consequentemente o desvio padrão da média de m observadores sobre o mesmo fluxo será $\sigma/\sqrt{m}$. Assim, quanto mais observadores, menor será o erro sobre o valor esperado.

É possível realizar estas observações independentes utilizando m funções *hash* diferentes, mas esta abordagem introduziria um grande custo computacional. Na prática, a técnica mais utilizada é dividir o multiconjunto em m subconjuntos e realizar as observações em paralelo, utilizando apenas uma função *hash* e computando o valor agregado pela média no final.

Neste trabalho iremos focar no algoritmo *HyperLogLog*, por ser um dos mais difundidos atualmente, além de ser o algoritmo a alcançar a representação mais compacta (para o mesmo erro relativo) dentre os citados.

2.4.1 Definição

O algoritmo *HyperLogLog* se baseia na observação do padrão de bits resultante da aplicação de uma função *hash* sobre os elementos do conjunto.

A função *hash* utilizada no *HyperLogLog* mapeia cada elemento do domínio do conjunto uniformemente para $\{0,1\}^\infty$, isto é, no conjunto de cadeias binárias distintas de tamanho infinito. Com isso, para determinado elemento x, a probabilidade de que o i-ésimo bit do hash de x seja 0 ou 1 é exatamente $^1/_2$, para todo $i \geq 1$. Assim, podemos também calcular a probabilidade de um valor *hash* possuir certo prefixo. Em especial, estaremos interessados na probabilidade de um prefixo que comece com um certo número de 0's, a saber:

$$\begin{aligned}
\Pr(h(x) = 1...) &= 2^{-1} \\
\Pr(h(x) = 01...) &= 2^{-2} \\
\Pr(h(x) = 001...) &= 2^{-3} \\
&\vdots \\
\Pr(h(x) = 0^{p-1}1...) &= 2^{-p}
\end{aligned}$$

Ao observar-se que, sobre todos os *hashes* de elementos do conjunto, o prefixo que possui maior número de 0's iniciando a cadeia é da forma $0^{p-1}1$ num fluxo, deduz-se que o número de tentativas esperadas para encontrar este valor é igual a 2^p. É importante notar como este conceito é análogo àquele de manter o menor valor *hash* encontrado, do algoritmo *MinCount* [Gir09].

O algoritmo consiste em particionar o conjunto aleatoriamente em $m = 2^b$ subconjuntos, usando os b primeiros bits do *hash* de cada elemento e armazenar, para cada subconjunto, o tamanho do maior prefixo $0^{p-1}1$. Estes valores são armazenados em um vector M de m posições. Cada posição do vetor representa um estimador e pode assumir um valor entre 0 e $\lceil \log_2 n \rceil$.

A estimativa da cardinalidade é feita computando a cardinalidade provável de cada subconjunto e obtendo a média harmônica entre todos os valores. Isto é:

$$\hat{n} = \alpha_m m^2 \left(\sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$$

A constante α_m é introduzida para corrigir um viés multiplicativo sistemático inerente ao cálculo da média harmônica de potências de 2 e é definida como:

$$\begin{aligned} \alpha_m &= \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1} \\ &\approx 0,7213/(1+1,079/m), \text{ para } m \geq 128 \end{aligned}$$

O uso da média harmônica é o grande diferencial do *HyperLogLog* em relação ao seu predecessor *LogLog*. Esta exige uma demonstração teórica mais complexa, porém diminui a variância dos estimadores individuais e permite melhorar a precisão do algoritmo em cerca de 20% [FFGM08]. Esta é uma mudança similar à sugerida no trabalho de Chassaing e Lucas [CG07], que aplica o mesmo tipo de melhoramento ao algoritmo *MinCount*. O Algoritmo 20 resume o que foi apresentado até agora.

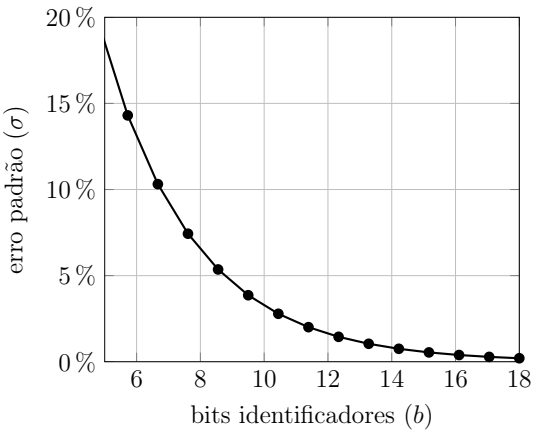
É importante observar que se a estimativa E encontrada estiver no intervalo $(\frac{5}{2}m, \frac{1}{30}2^{32}]$, o algoritmo a considera o retorno apropriado. Entretanto, se $E \leq \frac{5}{2}m$, a variância dos estimadores pode causar um erro maior que o esperado. Para este caso, utiliza-se o algoritmo *Linear Counting* [WVZT90] sobre o vetor M. Se $E > \frac{1}{30}2^{32}$, pode haver muitas colisões entre as funções *hash*. Para corrigir o valor estimado, o algoritmo compensa as possíveis colisões de forma análoga ao caso anterior, mas considerando E como o número de *hashes* distintos vistos de um total possível de 2^{32}.

O erro esperado para o *HyperLogLog* depende apenas da quantidade m de posições no vetor M. O algoritmo é capaz de produzir estimativas com erro padrão de $1,04/\sqrt{m}$. Isto é, para $b = 11$, i.e., $m = 2048$, o erro padrão esperado é de 2,3%. A Figura 23 mostra o erro padrão por número m de posições no vetor M.

Algoritmo 20 Estima a cardinalidade do multiconjunto S

seja $\rho(w)$ o índice do primeiro bit 1 na representação binária de w
seja b o número de bits escolhidos para representar cada subconjunto e $m = 2^b$
seja $\alpha_m \approx 0,7213/(1 + 1,079/m)$, para $m \geq 128$
função ESTIMAR-CARDINALIDADE(S)
    $M[0..m-1] \leftarrow 0$
    para cada $e \in S$ **faça**
        $x \leftarrow h(e)$
        $j \leftarrow x_0 x_1 x_2 \cdots x_{b-1}$
        $w \leftarrow x_b x_{b+1} x_{b+2} \cdots$
        $M[j] = \max(M[j], \rho(w))$
    fim para
    $E \leftarrow \alpha_m m^2 \left(\sum_{j=0}^{m-1} 2^{-M[j]}\right)^{-1}$
    se $E \leq \frac{5}{2}m$ **então retorna** $m \ln(m/|\{0 \leq i < m | M[i] = 0\}|)$
    senão se $E \leq \frac{1}{30}2^{32}$ **então retorna** E
    senão retorna $-2^{32} \ln\left(1 - \frac{E}{2^{32}}\right)$
    fim se
fim função

Figura 23: Erro padrão por tamanho de M



Embora com um erro relativo maior que outros algoritmos, a grande vantagem do *HyperLogLog* é precisar de apenas $\log_2 \log_2 n$ bits por elemento em M. *HyperLogLog* tem um erro relativo 33% maior, para o mesmo número de posições em M, se comparado com o do algoritmo *Probabilistic Counting* ($0,78/\sqrt{m}$ [FM85]), por exemplo. Entretanto, em termos de representação em memória, *HyperLogLog* usa apenas 21% do número de bits requerido por *Probabilistic Counting*.

HyperLogLog também é facilmente paralelizável. É possível particionar o conjunto calculando cada parte isoladamente das demais. Posteriormente é possível realizar a união entre os resultados obtidos. A união entre dois *HyperLogLogs* de mesma dimensionalidade pode ser feita apenas obtendo o máximo em cada posição do vetor M.

A interseção entre dois *sketches* não é possível. É possível, porém, calcular a cardi-

nalidade da interseção entre dois conjuntos utilizando o princípio da inclusão-exclusão, usando a operação de união, que é facilmente computável.

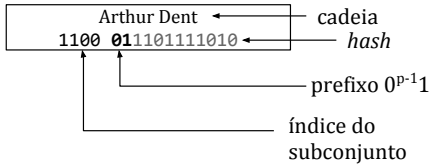
É importante notar que o erro estimado para esta operação é proporcional ao erro absoluto do maior conjunto sendo intersecionado. Isto pode levar a erros relativos extremamente altos.

É possível, entretanto, estimar a interseção de dois conjuntos através de seus *HyperLogLogs* computando um *MinHash* auxiliar, como discutiremos na Seção 2.4.4.

2.4.2 Exemplo

Com o objetivo de facilitar o entendimento da estrutura *HyperLogLog*, apresentamos aqui um exemplo prático de seu uso. Consideraremos aqui a simulação de inserção de cadeias de caracteres em uma estrutura *HyperLogLog* com $m = 16$ posições, usando uma função *hash* de 16 bits (por simplicidade). Como visto na seção anterior, não há garantias teóricas para $m < 128$, mas ignoraremos este fato por enquanto, apenas para fins didáticos. A Figura 24 explica como será representada cada entrada no exemplo.

Figura 24: Modelo de linha para exemplos



Cada linha (exceto a primeira) contém uma cadeia a ser inserida e o valor do *hash* 16 bits da cadeia, com os quatro primeiros bits em destaque. Esses bits representam o índice no intervalo $[0; 15]$ onde o resultado será inserido. Os 12 bits restantes são usados para computar o tamanho do prefiro $0^{p-1}1$ que será atualizado naquele índice. Além disso, na linha são representadas as 16 posições do vetor M com o estado após a inserção da cadeia representada na linha, destacando o índice referenciado. A primeira linha representa o estado inicial. A simulação pode ser vista na Figura 25.

A estimativa preliminar para o vetor M pode ser calculada de acordo com a fórmula descrita na seção anterior, isto é

$$\begin{aligned} E = \alpha_m m^2 (2^0 + 2^0 + 2^{-4} + 2^{-1} + 2^0 + 2^0 + 2^0 + 2^{-2} + \\ 2^0 + 2^{-1} + 2^0 + 2^0 + 2^{-2} + 2^0 + 2^0 + 2^{-3})^{-1} \\ = 0,6757304291820364 \times 256 \times 0,085561497 \approx 14,80 \end{aligned}$$

Como a estimativa preliminar é menor que $\frac{5}{2}m$, a estimativa final será dada pelo algoritmo LinearCounting, baseado no número de 0's no vetor M, isto é:

$$m \ln(m/10) \approx 7,52$$

Figura 25: Exemplos de inserção na estrutura *HyperLogLog*

Cadeias inseridas	M[0..m-1]															
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Arthur Dent **1100 01**1101111010	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**0**	**0**
Tricia McMillan **0011 1**00101101000	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**0**	**0**
Marvin **1111 01**0100110011	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**0**	**2**
Ford Prefect **0011 1**10010100000	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**0**	**2**
Zaphod Beeblebrox **1111 001**010111011	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**0**	**3**
Fenchurch **1001 1**01010110101	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**2**	**0**	**0**	**3**
Slartibartfast **0111 01**0101001000	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**2**	**0**	**1**	**0**	**0**	**2**	**0**	**0**	**3**
Prostetnic Vogon Jeltz **0010 0001**11110111	**0**	**0**	**4**	**1**	**0**	**0**	**0**	**2**	**0**	**1**	**0**	**0**	**2**	**0**	**0**	**3**

Após arredondamento, a resposta final do algoritmo, nesta simulação, seria 8, o mesmo número de cadeias inseridas na estrutura.

2.4.3 *HyperLogLog++*

Por se tratar de um problema muito recorrente na indústria, algoritmos para estimativa de cardinalidade começaram a ser amplamente utilizados e testados. Melnik et al. [MGL+10], ao descrever um motor para análise de dados, mostram o uso do algoritmo *MinCount* no Google.

Entretanto, Heule, Nunkesser e Hall publicaram em 2013 [HNH13] um artigo mostrando a melhor aplicabilidade do algoritmo *HyperLogLog* para os problemas de contagem do Google. Descrevem também uma melhoria que fizeram no algoritmo original para economizar o uso de memória e a mitigação do erro em casos de baixas cardinalidades. O algoritmo, com essas mudanças, é conhecido como *HyperLogLog++*.

As mudanças, embora simples, trazem considerável ganho em aplicações práticas:

Usar uma função *hash* de 64 bits

Com esta mudança, é possível estimar cardinalidades de conjuntos com mais de 2^{32} elementos. Além disso, torna-se obsoleto o trecho do algoritmo que lida com cardinalidades muito altas, pois a probabilidade de colisão é muito baixa para cardinalidades usuais.

Correção empírica no viés para baixas cardinalidades

A versão *prática* do algoritmo *HyperLogLog* original sugeria utilizar *Linear Counting* para baixas cardinalidades, para evitar um erro sistemático que ocorre na estimativa deste intervalo.

Heule et al. mostram que este erro era causado por um simples viés que pode ser medido empiricamente e corrigido, melhorando consideravelmente a precisão do

algoritmo. Além disso, demonstram empiricamente que ocorre uma anomalia no erro observado próximo à cardinalidade $\frac{5}{2}m$, definida no artigo original como ponto de transição entre a estimativa dada pelo *Linear Counting* e o *HyperLogLog* em si.

Na Seção 2.4.6, mostramos os efeitos empíricos desta correção em comparação com o algoritmo original.

Representação compacta de dados esparsos

Embora o algoritmo precise lidar com altas cardinalidades, na maior parte do tempo conjuntos com poucos elementos serão analisados. No algoritmo original, independente da cardinalidade do conjunto, a estrutura tem sempre o mesmo custo de memória. O artigo sugere uma representação mais compacta para estruturas esparsas. As principais mudanças envolvem guardar apenas os índices com valores no array utilizado pelo algoritmo, usar codificação de inteiros com tamanho variável e introduzir um passo de compressão para casos onde a estrutura é atualizada em lotes.

2.4.4 União e interseção

Em muitas situações pode ser útil computar *HyperLogLogs* de vários conjuntos e poder realizar operações entre eles. Por exemplo, como estimar quantas pessoas acessaram *o site A ou o site B*, tendo apenas o *sketch* dos usuários que acessaram cada site separadamente?

A operação mais simples de se calcular é a união entre conjuntos. A união entre conjuntos representados por dois *HyperLogLogs* de mesmo tamanho m consiste em obter o máximo entre cada uma das posições no vetor M. Em outras palavras,

$$M_{A\cup B}[i] = \max(M_A[i], M_B[i]) \text{ para todo } 0 \leq i < m.$$

Quando os dois *sketches* têm valores de m diferentes, é preciso diminuir o de maior tamanho sobre si mesmo quantas vezes forem necessárias para que fique com o mesmo tamanho do menor. Isto é:

$$M'_A[i] = \max(M_A[2i], M_A[2i+1]) \text{ para todo } 0 \leq i < \frac{1}{2}m.$$

A possibilidade de obter o *HyperLogLog* da união entre dois conjuntos torna o algoritmo facilmente paralelizável, pois é possível computar cada subconjunto em um nó de um cluster e apenas unir os resultados quando for necessário computar a cardinalidade.

No caso da interseção, não é possível fazer o mesmo. Não há na própria estrutura a informação suficiente para produzir o *sketch* relativo à interseção entre dois conjuntos. Uma alternativa é utilizar apenas a operação de união para estimar a cardinalidade através

do princípio de inclusão-exclusão.

$$|A \cap B| = |A| + |B| - |A \cup B|$$

Como todos os termos são estimáveis utilizando apenas operações já discutidas, é possível estimar a interseção. Entretanto, o erro absoluto é proporcional à cardinalidade da união entre os dois conjuntos. Se a interseção desejada for um conjunto pequeno, comparado com os conjuntos de entrada, o erro absoluto pode ser ordens de grandeza maior que a própria interseção.

Uma outra técnica, proposta por Pascoe [Pas13], consiste em manter um *MinHash* associado ao *HyperLogLog* estimar o índice de Jaccard sempre que for necessário computar a cardinalidade da interseção. A técnica se baseia na observação de que

$$|A \cap B| = J(A, B) \times |A \cup B|$$

pois

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

O erro desta técnica é limitado pelos erros das duas técnicas combinadas. Dados erros ϵ_H e ϵ_M associados às estimativas do *HyperLogLog* e *MinHash*, respectivamente, então

$$|A \cap B| \times (1 + \epsilon) = J(A, B) \times (1 + \epsilon_M) \times |A \cup B| \times (1 + \epsilon_H)$$

pode-se mostrar que

$$\epsilon = \epsilon_M + \epsilon_H + \epsilon_M \epsilon_H$$

Como o erro de nenhum dos dois algoritmos depende do tamanho do conjunto, podemos dizer que o erro da estimativa de interseção utilizando esta técnica também depende somente da quantidade de memória utilizada nos dois algoritmos.

2.4.5 Aplicações

O algoritmo *HyperLogLog* traz grandes vantagens tanto para aplicações em lote quanto em tempo real. Sua caracteristica altamente paralelizável o torna extremamente importante para aplicações que lidam com grandes volumes de dados. Abaixo citamos algumas aplicações comuns.

Bancos de dados: Um dos principais usos para algoritmos como *HyperLogLog* é a estimativa de cardinalidade em consultas de bancos de dados.

No Google, tanto *CountMin* quanto *HyperLogLog* são utilizados para responder consultas nos sistemas *Dremel* (sistema para análise de dados em larga escala) e *PowerDrill* (um banco de dados orientado a colunas). [HBB+12, MGL+10, HNH13].

O sistema de armazenamento Redis também permite o armazenamento de *sketches* de forma built-in [SH15], assim como o banco de dados PostgreSQL, que possui uma implementação nativa de *HyperLogLog* como uma das agregações de sua linguagem [CCD14].

A Amazon também anunciou a adição de uma agregação de contagem aproximada que utiliza HyperLogLog internamente [AWS15].

Segurança de infraestrutura: Uma dos tipos de sistemas que mais se beneficia de algoritmos de *streaming* são os que cuidam da segurança de infraestruturas, pois precisam processar em tempo real os dados que entram e saem das aplicações para detectar possíveis ataques.

Chabchoub, Chiky e Dogan [CCD14] descrevem um método para detectar um tipo de ataque de varredura de portas utilizando *HyperLogLog*. A ideia seria utilizar uma variante do algoritmo que usa uma janela deslizante para contar elementos distintos [CH10], verificando situações onde o número de portas distintas acessadas a partir de um roteador crescesse abruptamente num período de tempo curto.

O serviço OpenDNS utiliza *HyperLogLog* para detectar *malwares* que criam múltiplos nomes de domínio apontando para um conjunto pequeno de endereços IP [Den13]. Em vez de manter todos os nomes de domínio que resolveram para um certo IP, eles mantém apenas um *sketch* que conta quantos domínios distintos aos quais o IP corresponde.

2.4.6 Resultados Experimentais

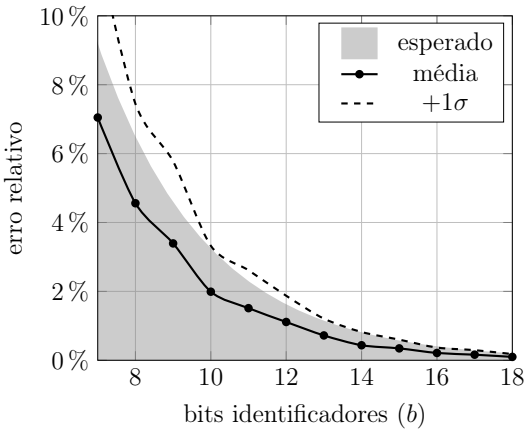
A fim de observar a previsões teóricas descritas nas seções anteriores, conduzimos três experimentos. O objetivo principal de cada um deles foi, respectivamente:

1. observar o erro na estimativa de conforme cresce o valor de b no algoritmo;
2. comparar as variantes sem correção de viés, com correção e *HyperLogLog++* ao longo do ponto crítico (onde os algoritmos trocam de técnica de estimativa) e
3. observar o erro ao estimar a cardinalidade da interseção usando *HyperLogLog* com *MinHash*.

No primeiro experimento, utilizamos o conjunto de obras de Shakespeare (42 no total) e estimamos a cardinalidade das palavras distintas em cada uma delas, comparando com a cardinalidade real e registrando a média e o desvio padrão do erro para cada valor de b diferente ($7 \leq b \leq 18$). A Figura 26 mostra o resultado deste experimento.

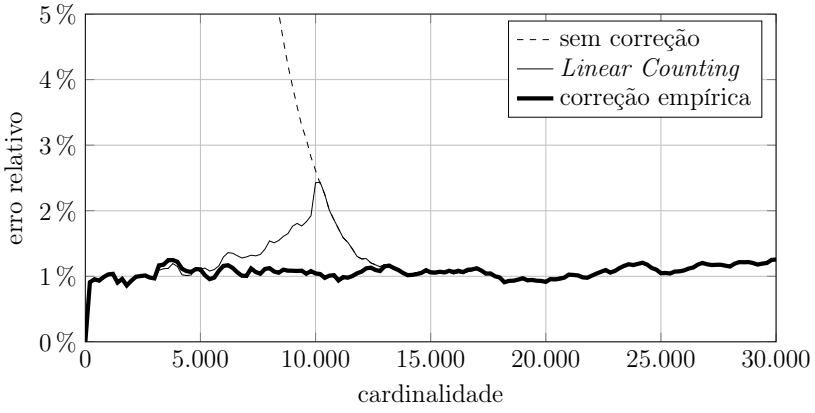
No segundo experimento, como o objetivo era testar a cardinalidade próxima ao ponto crítico, era necessário estimar cardinalidades maiores. Para isso, foram geradas 30 mil

Figura 26: Erro relativo observado



cadeias aleatórias de dez caracteres e inseridas num *HyperLogLog* com $b = 12$. Para esta configuração, o primeiro ponto crítico ($n = \frac{5}{2}m$) estava próximo a $n = 10240$. O experimento foi executado 64 vezes e a média do erro foi registrada para cada variante do algoritmo. A Figura 27 mostra o resultado deste experimento.

Figura 27: Erro relativo de versões do HyperLogLog para $b = 12$.



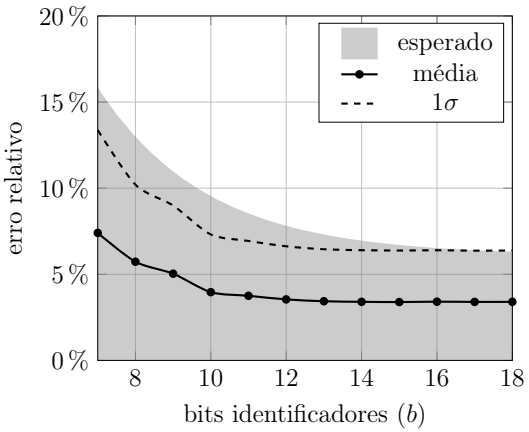
É possível observar a curva próxima ao ponto crítico, onde tanto o algoritmo *Linear Counting* quanto o *HyperLogLog* original levam a um erro acima do esperado. Também é possível observar como o *HyperLogLog++* diminui este erro através da correção com fatores obtidos empiricamente.

O terceiro experimento tinha como objetivo observar o erro da estimativa de cardinalidade entre a interseção de dois conjuntos. Para isso, usou-se o mesmo conjunto de dados utilizado no experimento com o *MinHash* (Seção 2.3.8): 84 conjuntos, sendo 42 deles versões com um número aleatório de palavras alteradas dos 42 conjuntos originais, totalizando 3486 pares de conjuntos. Para cada par, estimou-se a cardinalidade da interseção usando a técnica descrita na Seção 2.4.4 e comparou-se com a cardinalidade real.

O *MinHash* em todos os testes utilizava a técnica com apenas uma função *hash* e $k = 2048$. O *HyperLogLog* variava com $7 \leq b \leq 18$. O erro para cada estimativa foi

registrado e, para cada valor de b, foi computada a média e o desvio padrão. O resultado pode ser visto na Figura 28.

Figura 28: Erro relativo observado



Pelo resultado é possível observar como existe um certo ponto onde o erro do *MinHash* passa a dominar a estimativa, e a partir dele não vale mais a pena aumentar a precisão do *MinHash*. Para $k = 2048$, este valor mostrou-se próximo a $b = 14$, com um erro médio de aproximadamente $3,5\%$.

2.5 Considerações sobre as estruturas

Neste capítulo, apresentamos quatro estruturas de dados importantes para a representação probabilística de conjuntos. Cada uma delas permite certas operações sobre os conjuntos que representam. Nesta seção, destacaremos as similaridades e diferenças entre essas estruturas. A Tabela 6 revisa a tabela apresentada no início do capítulo, que resume as estruturas apresentadas nas seções anteriores, incluindo os parâmetros e o cálculo do limite superior do erro descritos ao longo do capítulo.

2.5.1 Filtro de Bloom e *Count-Min sketch*

É possível perceber uma certa semelhança na construção das estruturas filtro de Bloom e *Count-Min sketch*. Além disso, são as únicas estruturas dentre as quatro que possuem complexidade de espaço linear. De fato, *Count-Min sketch* é descrita na literatura como uma variante de um filtro de Bloom, conhecida como *Spectral Bloom filter* [CM03]. É possível, entretanto, prever a probabilidade de erro (falsos positivos) no filtro de Bloom com certa segurança. No caso de *Count-Min sketch* é preciso recorrer à desigualdade de Markov para definir um limite superior no erro da estimativa da frequência de elementos específicos. Por este motivo, seu erro teórico é superestimado. Por outro lado, *Count-Min sketch* permite outras operações quando usada para representar vetores: é possível

Tabela 6: Sumário de estruturas abordadas neste capítulo

Estrutura	**Parâmetros**	**Consulta**	**Erro**
Filtro de Bloom	m bits k funções *hash*	Pertinência	$\Pr[\textsc{FalsoPositivo}] \approx 0,6185^{m/n}$
		Cardinalidade	$\sigma_{\hat{n}} = \frac{\sqrt{m(e^{n/m}-(n/m)-1)}}{n}$
Count-Min sketch	$m = \lceil e/\epsilon \rceil$ $k = \lceil \ln(1/\delta) \rceil$	Frequência	$\Pr\left[\widehat{A[i]} > A[i] + \epsilon\|A\|_1\right] \leq \delta$
		Produto escalar	$\Pr\left[\widehat{A \cdot B} > A \cdot B + \epsilon\|A\|_1\|B\|_1\right] \leq \delta$
		Somatório de intervalos	$\Pr\left[\widehat{Q(a,b)} > Q(a,b) + 2\epsilon \log_2 n\|A\|_1\right] \leq \delta$
MinHash	$k \geq \frac{2+\theta}{\theta^2} \times \ln(2/\delta)$	Similaridade	$\Pr\left[\left\|\frac{J(\hat{A},B)}{J(A,B)} - 1\right\| > \theta\right] \leq \delta$
HyperLogLog	$m \times \lceil \log_2 \log_2 n \rceil$ bits	Cardinalidade	$\sigma_{\hat{n}} \approx 1,03896/\sqrt{m}$

estimar também o produto escalar usando a mesma estrutura e o somatório de intervalos de elementos no vetor com uma simples adaptação na representação.

2.5.2 *MinHash* e *HyperLogLog*

MinHash e *HyperLogLog* são menos "versáteis" se comparadas às outras estruturas, porém ambas possuem complexidade de espaço sublinear. São especializadas em estimar um parâmetro apenas: similaridade de Jaccard entre conjuntos, no caso do *MinHash* e cardinalidade de elementos distintos em multiconjuntos, no caso do *HyperLogLog*. Ambas as estruturas baseiam-se na definição de um estimador para a métrica que representam e na composição desses estimadores para redução da variância. No caso de *MinHash*, o estimador é definido pelo menor resultado de uma função *hash* aplicada a cada elemento do conjunto; a probabilidade de dois conjuntos terem o mesmo menor elemento é igual à similaridade de Jaccard entre os dois. No caso do *HyperLogLog*, o estimador é definido pelo maior prefixo $0^{p-1}1$ visto na representação binária do resultado de uma função *hash* aplicada a cada elemento do multiconjunto; a estimativa da cardinalidade de elementos distintos é dada por 2^p.

Há sinergia entre *MinHash* e *HyperLogLog* também para estimar a cardinalidade da interseção de conjuntos. Dado que, para dois conjuntos A e B, *MinHash* permite estimar $\frac{|A \cap B|}{|A \cup B|}$ e *HyperLogLog* permite estimar $|A \cup B|$, combinando as duas estruturas é possível estimar a cardinalidade $|A \cap B|$ com erro relativo apenas ao tamanho da interseção dos conjuntos.

3 REPRESENTAÇÃO PROBABILÍSTICA DE GRAFOS

Estruturas de dados probabilísticas apresentam novas formas de resolver problemas clássicos sob um ponto de vista probabilístico. Apresentaremos, neste capítulo, ideias para representações probabilísticas de grafos utilizando estruturas descritas nos capítulos anteriores.

Spinrad mostra em [Spi03] que $\Theta(n^2)$ bits são necessários para representar qualquer grafo através da clássica representação de grafos por matriz de adjacência. Classes específicas de grafos, entretanto, podem possuir representações mais compactas. Introduzimos a teoria sobre representações eficientes, bem como a definição de representações implícitas, ótimas ao longo da Seção 3.1.

Estruturas probabilísticas possuem grande aplicabilidade em bioinformática. Os resultados em [PHCK$^+$12], [ZPCK$^+$14], [OTM$^+$16] e [JB16] mostram o uso das estruturas probabilísticas discutidas ao longo deste trabalho em diversos passos na montagem de fragmentos de genoma bacterial. Apresentamos em detalhe a construção de uma representação probabilística baseada em filtros de Bloom para grafos *de Bruijn* na Seção 3.2.

Por fim, apresentamos novas ideias para representação implícita probabilística de grafos gerais, bem como de alguma subclasses específicas nas Seções 3.3 e 3.4.

3.1 Introdução a representações eficientes

Nesta seção, apresentamos conceitos sobre representações eficientes de grafos. Esta área baseia-se muito nos trabalhos de Muller [Mul88] e Kannan, Naor e Rudich [KNR92], entretanto, o livro de Spinrad [Spi03] sintetizou bem a teoria descrita até o momento, motivando novos trabalhos na área.

Para fins de notação, dizemos que um grafo G é denotado pela dupla (V, E), onde V é o conjunto de vértices, de cardinalidade $n = |V|$, e E é o conjunto de arestas, de cardinalidade $m = |E|$. Cada aresta é um par não-ordenado (u, v) com $u, v \in V$. A maior parte das discussões nesta seção aplicam-se a grafos rotulados (onde grafos isomorfos com rótulos diferentes são considerados grafos diferentes).

Começamos por discutir as duas representações clássicas de grafos: matriz de adjacência e lista de adjacência. Na discussão que se segue, está implícito o fato de que para nomear com cadeias binárias os elementos de um conjunto contendo n elementos, é necessário e suficiente utilizar cadeias de tamanho $O(\log n)$.

A matriz de adjacência utiliza $\Theta(n^2)$ bits para representar a presença ou ausência de cada uma das possíveis arestas entre dois vértices no grafo. A estrutura possui este nome pois consiste de uma matriz binária M de dimensão $n \times n$ onde $M[u, v] = 1$ se e somente

se os vértices u, v são adjacentes (assume-se $V = [1..n]$). A vantagem desta representação é a possibilidade de testar a adjacência entre dois vértices em tempo constante.

A lista de adjacência mantém, para cada vértice v, uma lista encadeada dos vértices u tal que $(u, v) \in E$. Para representar um grafo, n listas são mantidas, com um total de $2m$ itens em todas as listas (pois cada aresta é representada em exatamente duas listas). Cada item pode ser denotado por um natural no intervalo $[1; n]$ para representar o vértice e outro natural no intervalo $[1; 2m]$ para servir de apontador para o próximo item, precisando, portanto de $O(\log n + \log m) = O(\log n)$ bits para ser representado. Assim, a lista de adjacência representa um grafo de n vértices utilizando $O(m \log n)$ bits, assumindo um grafo conexo, no qual $m = \Omega(n)$.

Cada uma dessas representações tem vantagens e desvantagens. Por exemplo, não é possível testar a conectividade do grafo em tempo linear ($O(n+m)$) utilizando apenas uma matriz de adjacência. Com uma lista de adjacência seria possível fazê-lo, o que apresenta grande vantagem no caso de grafos esparsos. Entretanto o teste de adjacência nesta representação requer tempo no mínimo logarítmico, o que pode ser uma desvantagem em algoritmos intensivos em operações de teste de adjacência entre vértices arbitrários.

É possível, no entanto, analisar qual das representações é ótima em espaço para representação de grafos em geral. Uma representação é ótima se requer $O(f(n))$ bits para representar uma classe contendo $2^{\Theta(f(n))}$ grafos de n vértices.

Por exemplo, é possível provar que existem $2^{\Theta(n^2)}$ grafos com n vértices, pois há $n(n-1)/2$ arestas possíveis e cada grafo é uma combinação destas. Existem portanto, $2^{n(n-1)/2}$ grafos rotulados de n vértices, isto é $2^{\Theta(n^2)}$. O mesmo argumento serve para grafos não-rotulados, pois para cada grafo existem no máximo $n!$ isomorfismos, isto é, existem pelo menos $2^{n(n-1)/2}/n!$ grafos não-isomorfos. Como $n!$ é $2^{\Theta(n \log n)}$, então segue que o número de grafos não-isomorfos é $2^{\Theta(n^2)}$.

Desta forma, diz-se que a matriz de adjacência representa otimamente a classe contendo todos os grafos (rotulados ou não), pois requer $O(n^2)$ bits para representar uma classe contendo $2^{\Theta(n^2)}$ grafos. Já a lista de adjacência não é ótima, pois no caso de grafos completos, requer $\Theta(n^2 \log n)$ bits.

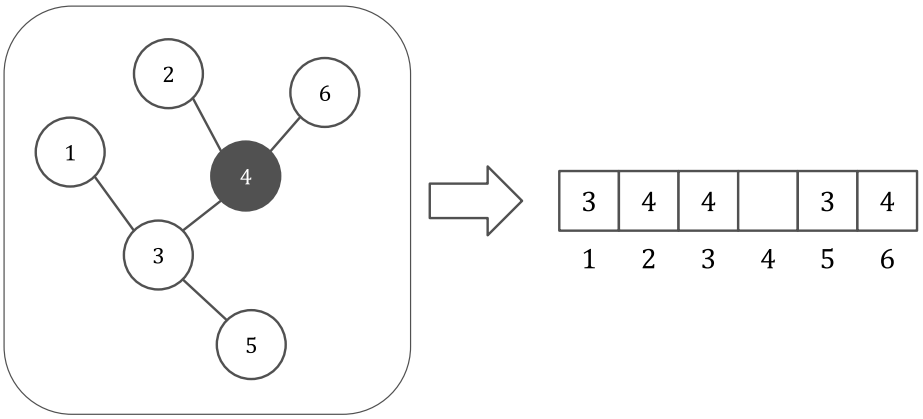
Muitas vezes, a representação escolhida altera a complexidade de certos problemas sobre o grafo que elas representam. Em [DGM02], Dahlhaus at al. introduzem uma representação derivada de uma lista de adjacência onde a lista relativa a cada vértice possui um bit que define se a lista representa as adjacências do vértice no grafo original ou em seu complemento. O trabalho mostra também que é possível computar diversos algoritmos sobre o complemento do grafo com tempo linear sobre a representação.

Para classes de grafos com $2^{O(n \log n)}$ elementos, uma representação ótima deve ter $O(n \log n)$ bits. Entretanto, apenas esta propriedade não é suficiente para garantir sua eficiência. Por exemplo, uma representação genérica ótima poderia ser definida enumerando todos os grafos em uma certa classe C, que possui $2^{\Theta(f(n))}$ elementos, e usar este

número que possui $\Theta(f(n))$ bits como representação. Entretanto, esta representação não permitiria o que teste de adjacência fosse realizado sem recriar o grafo original a partir da enumeração.

Estaremos em busca de representações ótimas que permitam o teste de adjacência em tempo constante. Em um exemplo prático, é trivial mostrar que existem $2^{O(n \log n)}$ árvores, pois sua representação como lista de adjacência usa $O(n \log n)$ bits (numa árvore, $m = O(n)$). Esta representação, entretanto, não favorece o teste de adjacência, pois a lista de cada vértice pode ter $O(n)$ itens. Uma representação mais apropriada seria definir um vértice arbitrário como raiz da árvore e manter, para cada vértice, apenas seu pai nesta arborescência. Assim, apenas $O(\log n)$ bits são necessários por vértice (para cada vértice, mantemos apenas o ponteiro para seu pai na árvore, se algum) e o teste de adjacência entre vértices pode ser feito de forma eficiente, apenas verificando se um dos vértices é pai do outro na representação. Um exemplo desta representação pode ser visto na Figura 29.

Figura 29: Exemplo de representação implícita de árvores (vértice raiz realçado)



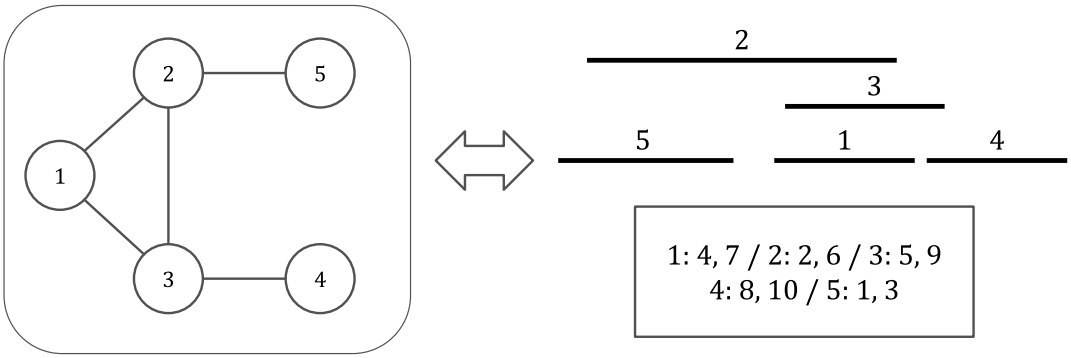
Esta *eficiência* do teste de adjacência parece estar ligada ao fato de a representação manter um número limitado de bits para cada vértice e utilizar apenas estes bits para o teste. Assim, podemos definir, motivados por essa intuição, o conceito de representação implícita [1] como a seguir. Seja C uma classe de grafos com $2^{\Theta(f(n))}$ elementos. Uma representação R de um grafo $G \in C$, de n vértices é dita implícita se:

1. *ela é assintoticamente ótima:* a representação requer apenas $O(f(n))$ bits no total;
2. *ela distribui informação entre os vértices:* a parcela de representação correspondente a cada vértice possui apenas $O(f(n)/n)$ bits;
3. *o teste de adjacência é local:* para testar a adjacência entre dois vértices, apenas as informações locais a eles são necessárias.

[1] Na literatura é usual definir como representação implícita apenas aquelas com $f(n) = n \log n$. A definição que apresentamos aqui é uma versão generalizada, formalizada em [Spi03], que é mais apropriada para os problemas que trataremos a frente.

Um exemplo de classe que permite representações respeitando essas propriedades são os *grafos de intervalo*. Um grafo é dito de intervalo se cada um de seus vértices puder ser mapeado para um intervalo na reta real, de modo que há aresta entre dois vértices se e somente se os respectivos intervalos possuem interseção não-vazia. Esta classe possui grande utilidade prática e pode ser definida uma representação implícita simples baseada na definição da classe. A representação consiste em enumerar as extremidades dos intervalos, na ordem em que aparecem na reta real, com inteiros no intervalo $[1; 2n]$. Representa-se cada vértice do grafo com os dois inteiros correspondentes às extremidades de seu respectivo intervalo. Um exemplo pode ser visto na Figura 30. Dois vértices serão considerados adjacentes se e somente se os intervalos representados pelos inteiros possuírem interseção não-vazia, o que pode ser testado em tempo constante. Apenas $\Theta(\log n)$ bits são usados para representar cada vértice e $\Theta(n \log n)$ bits são usados para representar o grafo inteiro. Isto indica que há $2^{O(n \log n)}$ grafos de intervalo possíveis.

Figura 30: Exemplo de representação implícita de grafos de intervalo



Para verificar o limite inferior do número de elementos na classe, considere grafos com n vértices onde cada um dos $n/2$ primeiros vértices possui uma aresta para um vértice distinto entre os $n/2$ últimos. Esta é uma subclasse dos grafos de intervalo. Por definição, ela possui $(n/2)!$ possíveis grafos com n vértices. Como, $(n/2)!$ é $2^{\Theta(n \log n)}$, segue que há $2^{\Theta(n \log n)}$ grafos de intervalo e, portanto, a representação apresentada anteriormente é ótima.

Encontrar uma representação que respeite as propriedades necessárias pode não ser trivial. De fato, para muitas classes pode ser que não exista representação implícita. Por exemplo, considere a classe de grafos onde $m = O(n)$. Usando uma lista de adjacência, é possível representar grafos nesta classe usando $O(n \log n)$ bits, o que indica que esta classe possui $2^{O(n \log n)}$ elementos. É impossível, entretanto, satisfazer as propriedades (2) e (3) simultaneamente, pois é possível transformar um grafo G qualquer para um H nesta classe introduzindo n^2 vértices de grau zero. Assim, se houvesse uma representação implícita para H, seria possível representar G usando apenas $O(n \log n)$ bits. Isto implicaria que é possível representar qualquer grafo usando $O(n \log n)$ bits, o que é absurdo.

A possibilidade de construir grafos em classes com $2^{\Theta(n \log n)}$ grafos a partir de grafos gerais apenas adicionando vértices pode posar como um desafio para definir propriedades gerais sobre essas classes. Por isso, utilizam-se mais frequentemente classes hereditárias na busca por classes com representações implícitas. Uma classe é dita *hereditária* se para todo grafo G nesta classe, todo subgrafo induzido de G também está na classe. A classe definida anteriormente (onde $m = O(n)$) não é hereditária, pois a remoção de vértices de grafos naquela classe pode resultar em grafos fora dela. É possível provar que uma classe hereditária C contém $2^{\Theta(n^2)}$ grafos de n vértices se e somente se ela contém todos os grafos bipartidos, todos os co-bipartidos ou todos os grafos *split*. É uma conjectura aberta se toda classe hereditária com $2^{O(n \log n)}$ grafos possui uma representação implícita. Essa conjectura é conhecida como *Conjectura da Representação Implícita* [KNR92, Spi03, Cha16]. É importante notar que mesmo classes não-hereditárias podem possuir representação implícita (por exemplo, árvores).

A fim de estudar o uso de estruturas de dados probabilísticas no problema de representações eficientes de grafos, introduzimos aqui a definição de *representação probabilística* como uma representação que relaxa o teste de adjacência, permitindo uma taxa fixa de falsos positivos e falsos negativos independente do tamanho do grafo (uma taxa de erro zero significa que a representação é em particular determinística).

Definimos ainda a ideia de representação implícita probabilística, que estende a definição apresentada anteriormente para permitir a aplicação de representações probabilísticas. Logo, uma representação é implícita probabilística se ela respeita as três propriedades de representações implícitas, enquanto é ao mesmo tempo probabilística.

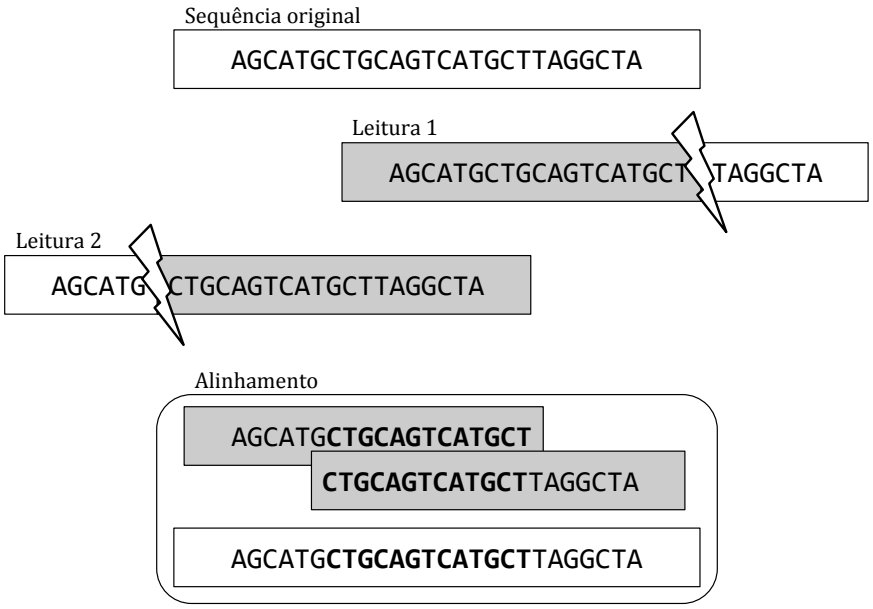
3.2 Representações probabilísticas: uma aplicação

Como visto na seção anterior, não é possível representar deterministicamente uma classe com $2^{\Theta(f(n))}$ grafos de n vértices usando $o(f(n))$ bits. Entretanto, há situações onde o limite pode não ser viável na prática ou, mesmo sendo viável, necessite de estruturas especiais para alcançar um uso de recursos aceitável.

Como exemplo motivador, analisaremos o problema da montagem de fragmentos de DNA – formados pelas bases nitrogenadas Adenina (A), Citosina (C), Guanina (G) e Timina (T) –, sequenciados através do método *shotgun*. Neste método, são sequenciadas somente cadeias curtas (entre cem e mil pares de bases). Cadeias mais longas são subdivididas em fragmentos, que precisam ser remontados posteriormente. A Figura 31 exemplifica o processo.

Este tipo de sequenciamento é essencial para o entendimento da composição microbial de amostras obtidas diretamente do ambiente (metagenômicas) que, em conjunto, desempenham importante papel no equilíbrio bioquímico de seu ambiente de origem. Mas, para permitir uma remontagem com alto grau de confiança, é preciso realizar, armazenar e

Figura 31: Processo de remontagem de sequências a partir de fragmentos



processar um número muito grande de leituras. Somente recentemente o equipamento necessário para realizar estas leituras começou a tornar-se acessível. Entretanto, os algoritmos tradicionais para a remontagem dos fragmentos não lidam satisfatoriamente com o imenso volume de dados gerados no processo.

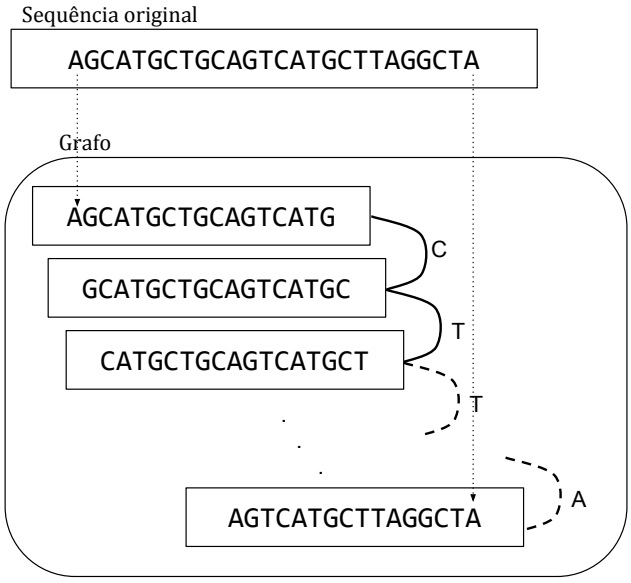
Para fins de processamento, as leituras são decompostas em palavras de um certo tamanho fixo k – chamadas de k-mers – e organizadas como um subgrafo induzido de um grafo de *de Bruijn* $(4, k)$ [PTW01, CPT11]. Um grafo de *de Bruijn* (m, n) possui m^n vértices, representando cada uma das cadeias de n caracteres formadas a partir de um alfabeto de m símbolos, e há aresta entre dois vértices se eles compartilham $n - 1$ caracteres contíguos. No grafo definido, cada vértice representa um k-mer e há aresta entre dois vértices se os k-mers relativos a eles compartilham $k - 1$ bases contíguas em alguma leitura. A Figura 32 mostra um exemplo desta construção.

É possível perceber que, nesse grafo, cada vértice pode ter no máximo oito arestas, uma para cada base possível (G, T, C e A) em cada uma das extremidades do k-mer. Espera-se também que a maioria dos vértices tenha duas arestas. Além disso, a escolha de k define o número máximo de vértices do grafo, dado por 4^k. Isto influencia também a probabilidade de encontrar arestas espúrias (que não representam um trecho real de sequência genômica).

O processo de montagem consiste em encontrar um caminho Hamiltoniano neste grafo. Em [PTW01] é mostrada uma construção alternativa que permite efetuar a montagem através da busca por um caminho Euleriano.

Para amostras de um mesmo indivíduo, os métodos tradicionais ainda conseguem representar e processar os dados utilizando *hardware* amplamente disponível. Gnerre et al. [GMP+11] mostram que é possível realizar a montagem de DNA humano, a partir

Figura 32: Grafo de *de Bruijn* gerado a partir de 16-mers de uma única sequência



de cerca de 3×10^9 leituras, utilizando 512GB de RAM. Este custo é bastante viável, especialmente considerando os recursos computacionais de provedores comerciais de serviços de nuvem. Entretanto, para amostras metagenômicas complexas, muitos terabytes de memória seriam necessários para representar todas as sequências de diferentes espécies em um mesmo grafo. Como essas leituras representam múltiplos espécimes de espécies diferentes, a fim de diminuir o custo de processamento total, pode-se empregar uma técnica de particionamento, onde usa-se algum método menos custoso para dividir esse grafo em componentes (para cada espécie ou grupo de indivíduos) que podem ser processados separadamente em momentos diferentes.

Em [PHCK+12], os autores relatam o uso de filtros de Bloom para representar probabilisticamente o grafo inteiro, podendo, dependendo da configuração de falsos positivos, utilizar apenas 4 bits para representar cada k-mer. Esta representação é utilizada para percorrer o grafo identificando seus componentes e realizando o particionamento das leituras. A vantagem do filtro de Bloom neste caso é a impossibilidade de falsos negativos, que, se possíveis, poderiam particionar leituras de um mesmo espécime em componentes diferentes.

Na construção sugerida, as arestas não são armazenadas, apenas os vértices. Qualquer par de vértices cujos k-mers compartilhem $k-1$ bases contíguas são considerados adjacentes. O passo de uma busca no grafo consiste em partir de um certo k-mer e verificar a existência de todos os oito possíveis k-mers adjacentes em todo o grafo. Desta forma, o problema de representação é reduzido à representação probabilística do teste de pertinência no conjunto de k-mers. O filtro de Bloom é diretamente aplicável neste caso.

O argumento defendido pelos autores é que, em dados reais, a taxa de falsos positivos tende com maior probabilidade a causar uma elaboração local nos componentes, em vez

de causar conexões espúrias entre componentes. Argumenta-se também que através de análise experimental, probabilidades de falso positivo até cerca de 18% não costumam realizar mudanças significativas na macroestrutura do grafo. Assim, definindo um valor de $q = 4$ bits por elemento no filtro de Bloom, é possível alcançar uma taxa de falsos positivos de cerca de 15%, que é suficiente para o processo de particionamento das amostras metagenômicas.

3.3 Filtro de Bloom como representação implícita

Os resultados em [PHCK+12] suscitam a discussão sobre a viabilidade de representações probabilísticas para outras classes de grafos além dos *de Bruijn*.

Como exemplo, podemos estudar o uso direto de filtros de Bloom para representação do conjunto de arestas E em grafos gerais. Como o objetivo é representar grafos gerais, o desejável é que seja possível construir o filtro com complexidade de espaço igual a $O(m)$.

A ideia consiste em encontrar uma função *hash* $h : E \to [1..m_B]$, que mapeia arestas do grafo em posições em um filtro de Bloom de m_B bits. De fato, filtros de Bloom permitem representar toda a adjacência do grafo utilizando 10 bits por aresta – isto é, O(m) bits –, a fim de alcançar uma probabilidade de falsos positivos menor que 1%. Esta representação mostra grande valor para representação de grafos esparsos, apesar de ser igualmente eficiente à matriz de adjacência ao requerer $O(n^2)$ bits para representar o grafo no pior caso (ex.: grafos completos).

Esta representação possui uma característica importante: toda aresta do grafo é representada deterministicamente. Isto é, se a aresta existe no grafo original, com 100% de probabilidade ela existirá na versão probabilística. Desta propriedade, segue que se um teste de adjacência na representação probabilística resultar em resposta negativa, é garantido que a aresta não exista no grafo original. Isto é, como característica do filtro de Bloom, não há falsos negativos. O contrário, entretanto, não é verdade. Em testes de adjacência entre vértices não-adjacentes, há possibilidade de falsos positivos. Isto significa, que para uma probabilidade de falsos positivos p, espera-se encontrar $p(\frac{n(n-1)}{2} - m)$ arestas na representação probabilística que não existem no grafo original.

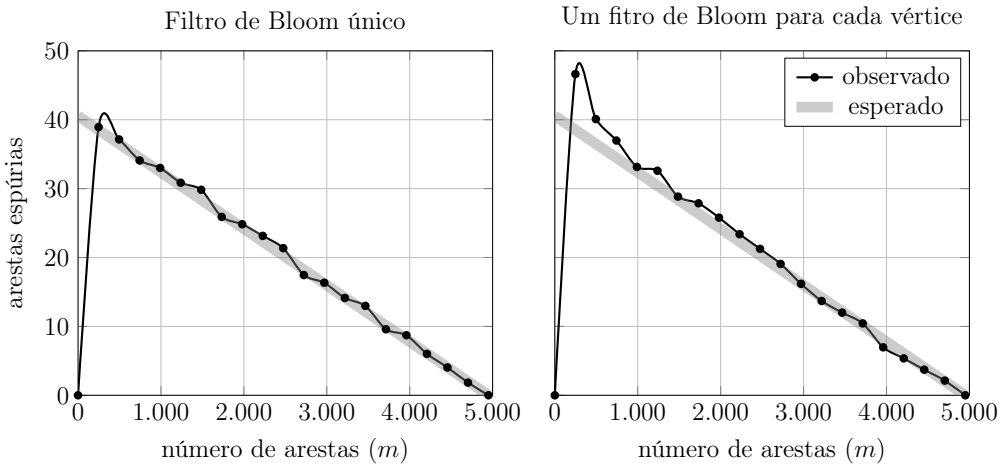
É importante notar que, construída da forma apresentada, esta não seria uma representação implícita, mesmo se relaxarmos o requisito do teste de adjacência para permitir respostas probabilísticas. Isto se deve a esta representação não distribuir a informação entre os vértices. De fato, não há representação local nos vértices.

Uma variação que resolveria este problema seria manter um filtro de Bloom para cada vértice. Assim, para cada vértice v, teríamos um filtro de Bloom com $10 \times \mathrm{d}(v)$ bits – onde $\mathrm{d}(v)$ é o grau do vértice – de forma que a probabilidade de falsos positivos em cada um deles é exatamente igual e equivalente à versão anteriormente apresentada. Assim, para cada vértice, esta representação necessita de $O(\mathrm{d}(v))$ bits que, no pior caso, é equivalente

a uma linha da matriz de adjacência ($O(n)$ bits). Entretanto, permite representação de grafos esparsos com complexidade menor que a lista associada a cada vértice numa lista de adjacência, que requer $O(\mathrm{d}(v) \log n)$ bits.

A Figura 33 compara os resultados empíricos para sucessivos testes em grafos aleatórios com $n = 100$ e m variando em todos os valores possíveis para um grafo com este tamanho. Ambas as variantes apresentadas nesta seção foram testadas. Note que para $m = 0$, por característica da construção filtro de Bloom, não há arestas espúrias.

Figura 33: Arestas espúras em um grafo com $n = 100$.



3.4 *MinHash* como representação implícita

Uma propriedade desejável de representações probabilísticas é que elas sejam capazes de representar grafos utilizando menos recursos que as melhores representações determinísticas. Nesta seção, apresentaremos algumas ideias de construções baseadas em funções *hash* sensíveis a localidade, em especial *MinHash* e *SimHash*, apresentadas anteriormente na Seção 2.3.

Essas estruturas funcionam sumarizando conjuntos através de assinaturas compactas que permitem, em tempo logarítmico ou menor, estimar um índice de semelhança entre os conjuntos que representam. No caso de *MinHash*, este índice é o coeficiente de *Jaccard* $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$. No caso do *SimHash*, o índice é relativo à similaridade de cosseno dos vetores que representam os conjuntos $S(A, B) = 1 - \arccos\left(\frac{|A \cap B|}{\sqrt{|A| \cdot |B|}}\right)/\pi$. Todos os métodos nesta seção podem ser usados com ambas as estruturas, porém, por brevidade, referenciaremos apenas *MinHash* ao longo do texto.

A ideia principal que trabalharemos nesta seção consiste em representar cada vértice como um conjunto, de forma que seja possível verificar a adjacência entre dois vértices comparando a semelhança entre os conjuntos que os representam. Isto é, dada uma família de conjuntos $\{S_1, \ldots, S_n\}$ e uma métrica de similaridade $f(S_i, S_j) \to \mathbb{R}$, e um limite infe-

rior δ, definimos o grafo como $(V = \{v_1, \ldots, v_n\}, E = \{(v_i, v_j) : 1 \leq i < j \leq n, f(S_i, S_j) > \delta\})$. Um problema que abordaremos adiante será o de como escolher tais conjuntos que representam um grafo. De antemão, porém, observamos que não é necessário que escolhamos uma família de conjuntos ótima no sentido de minimizar cardinalidade dos mesmos ou de sua união, por exemplo, pois os conjuntos em si não serão armazenados, somente suas assinaturas *MinHash*. Desta forma é possível estimar a adjacência entre dois vértices através da estimativa de similaridade entre seus conjuntos.

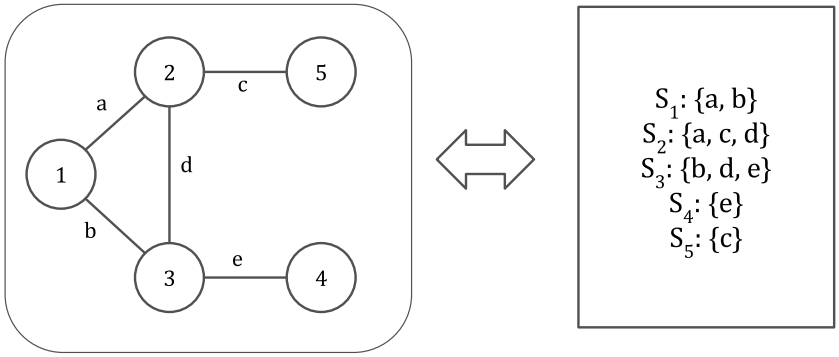
A vantagem desta abordagem vem das propriedades de *MinHash*. O erro relativo da estimativa de similaridade cresce em fator sublinear ao tamanho do conjunto, sendo mais significativa a quantidade de elementos na assinatura. Desta, forma, mesmo que o processo de construção dos conjuntos envolva a criação de conjuntos com cardinalidades polinomiais sobre o número de vértices do grafo, a assinatura *MinHash* que precisará ser armazenada para cada vértice irá conter apenas um número constante de elementos.

3.4.1 Generalizando grafos de interseção

A ideia de representar a adjacência de vértices em um grafo através de conjuntos remete diretamente à ideia de grafos de interseção. Um grafo de interseção de n vértices é definido através de uma família de conjuntos $\{S_1, \ldots, S_n\}$, de forma que há adjacência entre v_i e v_j se e somente se $S_i \cap S_j \neq \varnothing$. Usando notação apresentada anteriormente, é possível definir também através da função de similaridade $f(S_i, S_j) = J(S_i, S_j) = \frac{|A \cap B|}{|A \cup B|}$ e $\delta = 0$.

É fácil observar que todo grafo é um grafo de interseção. Por exemplo, é possível definir S_i como o conjunto de todas as arestas incidentes em v_i. Assim, a interseção $S_i \cap S_j$ irá conter uma aresta se e somente se v_i e v_j compartilharem uma aresta. A Figura 34 demonstra esta construção.

Figura 34: Exemplo de grafos de interseção



Apresentamos aqui uma construção generalizada para outros valores de δ, em especial, para qualquer $\delta \in \mathbb{Q} \cap [0; 1]$. Dado um grafo $G = (V, E)$ qualquer, $\delta = \frac{x}{y}$, para x e y inteiros, o objetivo é construir conjuntos $\{S_1, \ldots, S_n\}$ tais que $(v_i, v_j) \in E$ se e somente se $J(S_i, S_j) > \delta$.

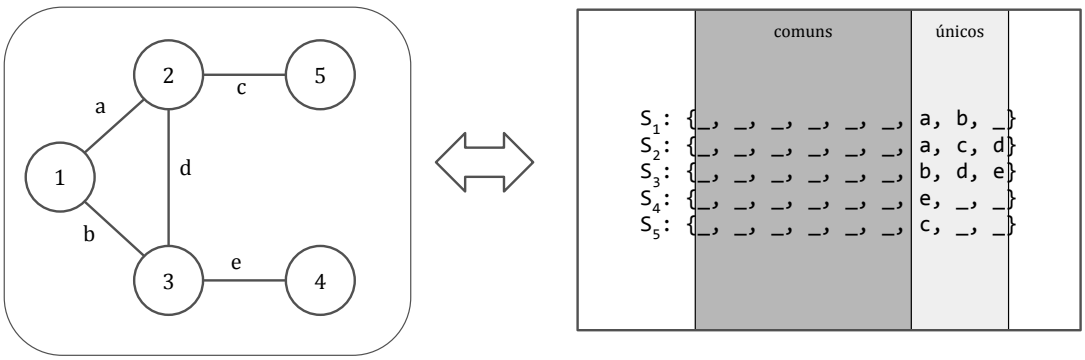
Para exemplificar a construção, começamos descrevendo um grafo com $E = \varnothing$. Neste grafo, todos os conjuntos são construídos para ter $J(S_i, S_j) = \delta$, ou seja, representando uma não-adjacência. Para tanto, cada conjunto é composto por a elementos comuns entre os conjuntos e b elementos únicos de cada conjunto. Desta forma, $|S_i \cap S_j| = a$ e $|S_i \cup S_j| = a + 2b$. Então, é possível determinar a proporção de a e b exata pela equação:

$$\frac{a}{a+2b} = \delta = \frac{x}{y} \Rightarrow a = \frac{2x}{y-x}b$$

Por esta relação, percebe-se que é suficiente escolher b de forma que seja divisível por $y - x$. Para modelar as adjacências, no conjunto de cada vértice v substitui-se $\mathrm{d}(v)$ dos b elementos únicos em cada conjunto pelas arestas incidentes no vértice relativo aquele conjunto. Portanto, também é preciso escolher $b \geq \Delta(G)$, onde $\Delta(G)$ é o maior grau de um vértice em G.

Por exemplo, para $\delta = \frac{1}{2}$, temos que $a = 2b$. A Figura 35 mostra um exemplo prático. Na figura, uma posição de elemento sem identificador significa que podemos escolher um elemento distinto qualquer. No exemplo, como $\Delta(G) = 3$, podemos assumir $b = 3$ e $a = 6$, logo, qualquer não-adjacência possui $J(S_1, S_2) = \frac{1}{2}$ e as adjacências possuem $J(S_1, S_2) = \frac{7}{12} > \frac{1}{2}$.

Figura 35: Exemplo de grafos de interseção para $\delta = 0,5$



De uma forma geral, a diferença entre os índices de similaridade que denotam adjacência e não-adjacência será igual a $(a + 2b)^{-1}$. Dado que $b \geq \Delta(G)$, essa diferença tende a tornar-se desprezível conforme o grafo cresce, se $\Delta(G) = \omega(1)$. Caso contrário, ela fica constante. Exemplos deste último caso são grafos com grau limitado, como o caso dos grafos k-regulares (para k fixo) e os grafos de *de Bruijn* para k-mers com k-fixo. Isto torna o uso desta construção com *MinHash* inviável para representar grafos de forma probabilística no caso geral, pois com o erro associado à estimativa, a incerteza sobre a adjacência ou não de dois vértices tornará o teste em grafos muito grandes indistinguível de uma resposta arbitrária.

Por este motivo, nas próximas seções, buscaremos apresentar construções que garantam um certo intervalo entre similaridades que representam adjacências e não-adjacências.

Mais específicamente, $\delta_A < \delta_B$ e:

$$(v_i, v_j) \notin E \leftrightarrow f(S_i, S_j) \leq \delta_A$$

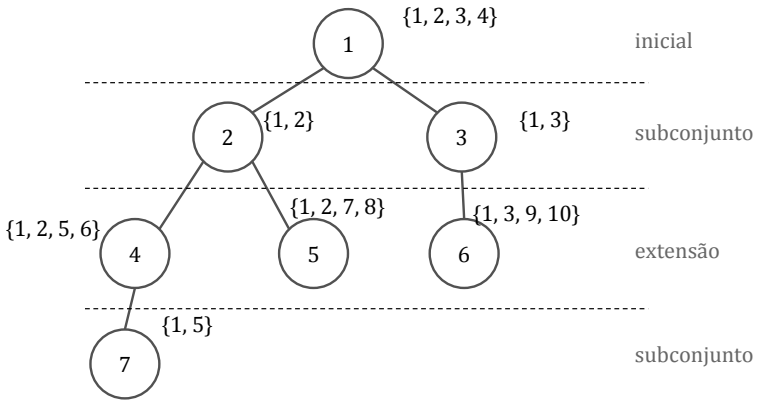
$$(v_i, v_j) \in E \leftrightarrow f(S_i, S_j) \geq \delta_B$$

Embora esteja claro que para $\delta_A = \delta$ e $\delta_B = \delta + \epsilon$ (onde ϵ é um valor não-negativo tão pequeno quanto necessário) existe construção viável para grafos gerais, como recém demonstrado, ainda é um problema aberto se para valores arbitrários de δ_A e δ_B esta construção sempre existe. Por exemplo, provaremos na Seção 3.4.3 que para $\delta_A = 0,4$ e $\delta_B = 0,6$, não há tal construção para grafos bipartidos. Por isso, abordaremos progressivamente o assunto para classes específicas de grafos de forma a explorar melhor as propriedades do problema.

3.4.2 Construção para árvores

Nesta seção, mostramos que é possível construir uma representação probabilística para árvores com complexidade de espaço menor do que a melhor construção determinística para $\delta_A = \frac{1}{3}$ e $\delta_B = \frac{1}{2}$. Esta construção é recursiva e baseia-se na ideia de que subconjuntos $S \subset U$, tais que $2|S| = |U|$, possuem índice de Jaccard $J(S, U) = \frac{1}{2}$. Então, partindo de um vértice arbitrário como raiz da árvore (portanto transformando a árvore em enraizada) com um conjunto inicial qualquer, define-se um procedimento para a escolha de subconjuntos de forma que, para dois filhos a e b quaisquer, seus subconjuntos S_a e S_b, respectivamente, respeitem $J(S_a, S_b) \leq \frac{1}{3}$. Os filhos desses vértices, por sua vez, podem ter conjuntos definidos pela extensão dos conjuntos de seus pais. A partir daí, o processo torna-se recursivo. A Figura 36 mostra um exemplo desta construção usando conjuntos de números inteiros.

Figura 36: Exemplo de construção de conjuntos para uma árvore



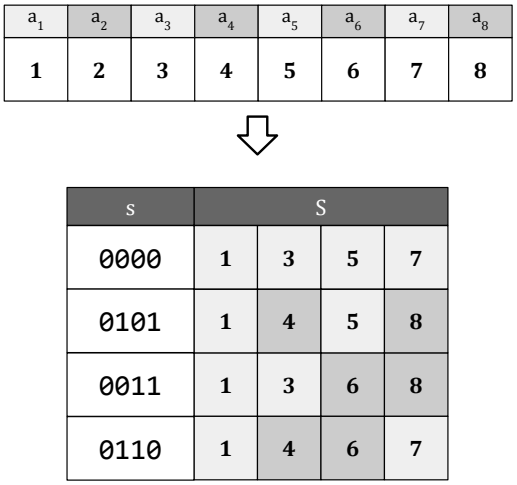
O número de elementos no conjunto inicial deve ser um valor x (divisível por quatro) suficiente para que seja possível extrair até $\Delta(G)$ subconjuntos de $x/2$ elementos, de forma

que nenhum par (S_a, S_b) de subconjuntos possua $J(S_a, S_b) > \frac{1}{3}$. Além disso, nunca devem ser escolhidos mais do que $x/4$ elementos do conjunto que tenham sido originados de um vértice ancestral, para garantir que o índice de Jaccard entre vértices e seus avôs seja sempre menor que $\frac{1}{3}$.

Qualquer estratégia de seleção de subconjuntos que tenha essas propriedades poderia ser usada. Oferecemos, como exemplo, uma estratégia capaz de selecionar, a partir de um conjunto com x elementos, $x/2$ subconjuntos de $x/2$ elementos, tais que quaisquer dois subconjuntos S_a, S_b possuam exatamente $x/4$ elementos em comum, ou seja, $J(S_a, S_b) = \frac{x/4}{3x/4} = \frac{1}{3}$.

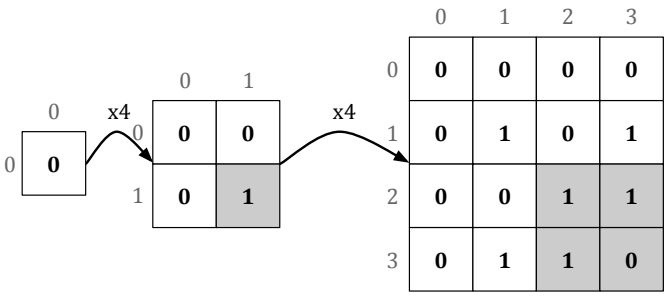
A ideia desta estratégia é gerar uma cadeia binária s, com $|s| = x/2$, que represente a seleção de elementos $S \subset \{a_1, \ldots, a_x\}$ de modo que se o i-ésimo bit de s tem o valor j, então a_{2i-1+j} pertence a S, como exemplificado a partir de um conjunto com oito elementos, na Figura 37.

Figura 37: Exemplo de subconjuntos para um conjunto inicial com oito elementos



A geração das cadeias binárias pode ser feita através de uma construção iterativa. Onde partindo de uma matriz 1×1, a cada passo a matriz é quadruplicada, sendo o quadrante inferior direito com os bits invertidos, como exemplifica a Figura 38.

Figura 38: Processo de construção de cadeias binárias



Esta construção garante que a cada passo o número de linhas duplique, mas cada

linha seja distinta das demais e possua exatamente metade de seus bits em comum com qualquer outra linha. Uma construção mais simples desta tabela vale-se do fato de que qualquer célula (i, j) nesta tabela é igual ao resto da divisão do número de bits 1 em `i&j` por dois, onde `&` denota o operador AND binário aplicado bit-a-bit.

Baseado nesta construção, podemos definir o algoritmo que gera os conjuntos para cada vértice da árvore. Ele pode ser visto no Algoritmo 21.

Algoritmo 21 Gera os conjuntos S_v para cada vértice v em uma árvore G

```
1: seja Δ(G) o maior grau de um vértice em G
2: seja q = 2^⌈log2(dmax(G))⌉+1
3: seja ρ(v, n) uma função que retorna n elementos únicos ao vértice v
4: função GeraSubconjunto(S = {s0, s2, ··· , sq−1}, i)
5:     R ← ∅
6:     para j ∈ [0, q/2) faça
7:         x ← BitCount(i&j)%2
8:         R ← R ∪ s2j+x
9:     fim para
10:    retorna R
11: fim função
12: procedimento VisitaFilhos(G, v)
13:    marcar v como visitado
14:    i ← 0
15:    para cada vizinho não-visitado u de v em G faça
16:        se |Sv| = q então
17:            definir Su = Gera-Subconjunto(Sv, i)
18:            i ← i + 1
19:        senão
20:            definir Su = Sv ∪ ρ(u, q/2)
21:        fim se
22:        VisitaFilhos(G, u)
23:    fim para
24: fim procedimento
25: procedimento GeraConjuntos(G)
26:    v ← um vértice qualquer em G
27:    definir Sv = ρ(r, q)
28:    GeraFilhos(G, r)
29: fim procedimento
```

Esta técnica permite representar probabilisticamente o grafo através de assinaturas com número constante de bits, alcançando uma representação probabilística com $O(n)$ bits (na versão com *SimHash*) para uma classe com $2^{O(n \log n)}$ grafos de n vértices. Portanto, esta representação possui complexidade de espaço menor do que a representação ótima da classe. Na prática, entretanto, para árvores com até cerca de 2^{64} vértices, a representação determinística continua sendo mais eficiente (por possuir um fator constante menor).

Para verificar a eficácia desta técnica de construção na prática, foram realizados ex-

perimentos com árvores aleatórias, variando alguns parâmetros, e as respectivas taxas de falsos positivos e falsos negativos foram registradas.

Considera-se falso negativo quando uma aresta é erroneamente representada como uma não-aresta, e falso positivo são o caso inverso, quando uma não-aresta é representada como uma aresta. Nos testes, ambos estão representados em valores relativos. A taxa de falsos positivos é a razão entre o número de falsos positivos e a quantidade de não-arestas no grafo. E vice-versa. Em tese, é possível ter tanto a taxa de falsos positivos como de falsos negativos iguais a 100%, mas isso não ocorre no teste em momento algum.

Tratando-se de uma árvore, um grafo esparso, os falsos negativos geralmente são menos desejáveis do que falsos positivos, mas de fato qual dos dois é mais ou menos aceitável depende da aplicação. A construção que apresentamos nesta seção gera uma representação *MinHash* com $\delta_A = \frac{1}{3}$ e $\delta_B = \frac{1}{2}$. O valor da similaridade computada, entretanto, pode estar entre δ_A e δ_B devido ao erro associado à estrutura probabilística. Portanto, a verificação de adjacência nesta representação depende da escolha de um limite após o qual os valores de similaridades passam a ser considerados arestas.

O primeiro teste analisa este limite, que por analogia também chamaremos de δ. Isto é, dois vértices v_i e v_j são considerados adjacentes se, dada a função probabilística de similaridade f, $f(S_i, S_j) > \delta$. No teste, variamos os valores de δ, com árvores aleatórias de 200 vértices, para $k = 64$ e $k = 128$ e observamos as curvas geradas pelas taxas de falsos positivos e falsos negativos. O resultado pode ser visto na Figura 39.

O segundo teste analisa a evolução do percentual de falsos positivos e falsos negativos com o crescimento do grafo. Neste teste usamos $k = 128$ e $\delta = \frac{5}{12}$. No caso do SimHash, uma transformação é necessária, pois a estrutura não representa o índice de Jaccard diretamente. Portanto, o valor utilizado foi $\delta = 1 - \frac{\arccos(5/12)}{\pi} \approx 0,671173911$. O resultado pode ser visto na Figura 40. Como o erro das estruturas não depende do tamanho do conjunto, não era esperado que o erro variasse consideravelmente com diferentes grafos.

É importante notar, entretanto, a diferença entre os testes com *MinHash* e *SimHash*. No primeiro, o erro é predominante de falsos positivos, enquanto no segundo, há um balanço maior entre falsos positivos e falsos negativos. Isso pode ser devido ao comportamento da curva vista no teste anterior.

3.4.3 Considerações sobre grafos bipartidos

Embora a construção apresentada na seção anterior permita a representação probabilística de árvores com complexidade assintótica menor que a melhor representação determinística, ela possui aplicabilidade prática apenas em situações onde um erro mais alto é aceitável ou em árvores muito grandes (da ordem de 2^{64} vértices).

Um resultado desejável seria uma construção polinomial de conjuntos para alguma classe de grafos com $2^{O(n^2)}$ elementos. Sabe-se que qualquer classe hereditária de grafos contém $2^{O(n^2)}$ se e somente se ela contém todos os grafos *bipartidos*, todos os *co-bipartidos*

Figura 39: Percentual de falsos positivos e falsos negativos por limite de teste.

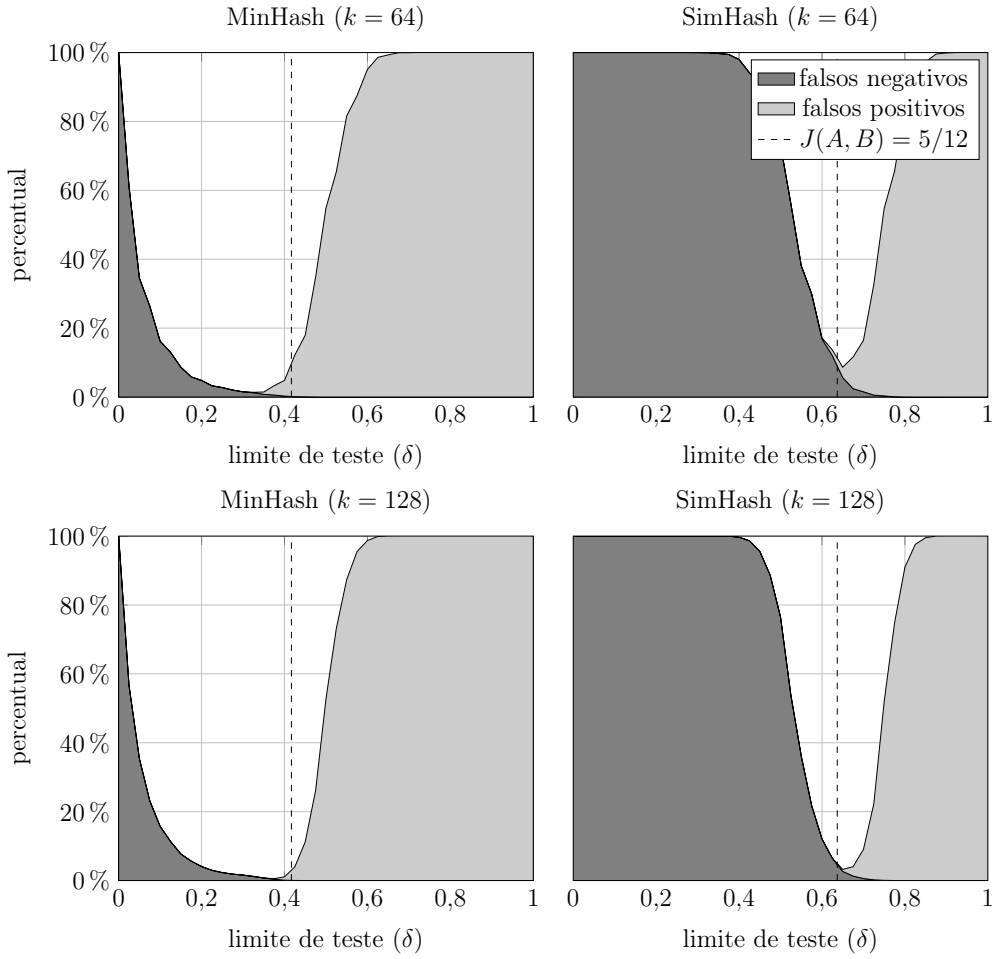
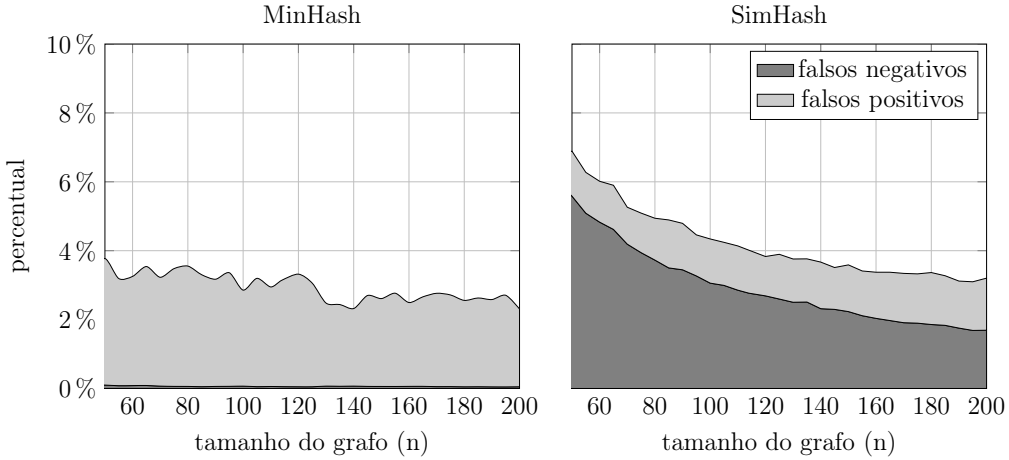


Figura 40: Percentual de falsos positivos e falsos negativos por tamanho do grafo.

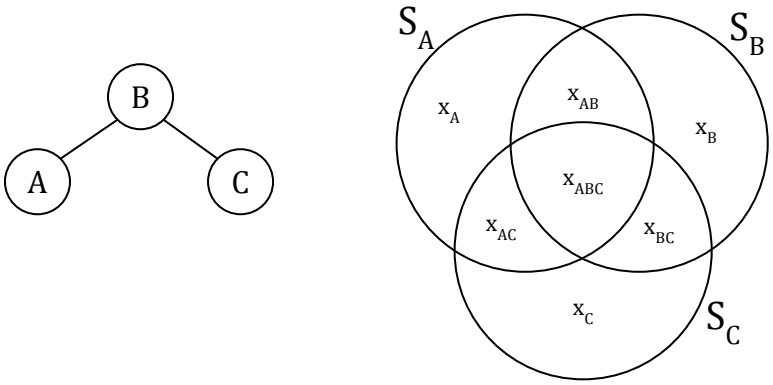


ou todos os *split* [Spi03]. Por isso, torna-se interessante o estudo de construção de conjuntos para essas classes.

Isto suscita a discussão sobre a viabilidade ou inviabilidade de representações probabilísticas baseadas em *MinHash* para todas as classes de grafos.

É possível demonstrar a inviabilidade de construção para certos valores de δ_A e δ_B em uma classe específica, bastando demonstrar que, para um grafo naquela classe não existe construção possível. Para tal, podemos modelar a busca por esses conjuntos como um problema de programação linear inteira e aplicá-lo a algum grafo específico da classe. A Figura 41 mostra como são definidas as variáveis do problema associado a um grafo com três vértices A, B e C.

Figura 41: Variáveis do problema para um grafo de três vértices



Definindo $\delta_A = 0,4$ e $\delta_B = 0,6$, é possível escrever o seguinte problema de programação linear inteira:

$$\begin{aligned}
\min \quad & x_A + x_B + x_{AB} + x_C + x_{AC} + x_{BC} + x_{ABC} \\
\text{sujeito a} \quad & 6x_A + 6x_B - 4x_{AB} + 6x_{AC} + 6x_{BC} - 4x_{ABC} \leq 0 \\
& -4x_A - 4x_{AB} - 4x_C + 6x_{AC} - 4x_{BC} + 6x_{ABC} \leq 0 \\
& 6x_B + 6x_{AB} + 6x_C + 6x_{AC} - 4x_{BC} - 4x_{ABC} \leq 0 \\
& x_A + x_{AB} + x_{AC} + x_{ABC} \geq 1 \\
& x_B + x_{AB} + x_{BC} + x_{ABC} \geq 1 \\
& x_C + x_{AC} + x_{BC} + x_{ABC} \geq 1
\end{aligned}$$

Cada variável $x_{S_1,\ldots,S_n}$ representa quantos elementos pertencem unicamente a $S_1 \cap \ldots \cap S_n$. A função objetivo é arbitrária e, assim sendo, escolhemos minimizar a quantidade total de elementos distintos sendo usados. Cada restrição está associada a um par x, y de vértices distintos do grafo. Caso eles sejam adjacentes, queremos que $J(x, y) \geq \delta_B$; caso contrário, que $J(S_x, S_y) \leq \delta_A$. A restrição surge diretamente da substituição das expressões de interseção e união implícitas em $J(S_x, S_y)$ pelas somas convenientes das variáveis. Por exemplo, a primeira restrição está associada aos vértices A e B. Como eles são são adjacentes, eles devem cumprir

$$\frac{x_{AB} + x_{ABC}}{x_A + x_B + x_{AB} + x_{AC} + x_{BC} + x_{ABC}} \geq \frac{6}{10}$$

que resulta na restrição fornecida.

Uma solução deste problema se dá com $x_{AB} = 1$, $x_{BC} = 1$ e $x_{ABC} = 1$, mostrando que uma construção para os parâmetros dados poderia ser, por exemplo, $S_A = \{1, 3\}$, $S_B = \{1, 2, 3\}$ e $S_C = \{2, 3\}$.

Ao aplicar a mesma ideia para o grafo bipartido completo $K_{3,3}$, para $\delta_A = 0,4$ e $\delta_B = 0,6$ (construção omitida por brevidade), é possível verificar que o problema não possui solução viável, demonstrando portanto a inviabilidade, para estes parâmetros da representação de grafos bipartidos. O mesmo ocorre em $K_{4,4}$, para $\delta_A = \frac{1}{3}$ e $\delta_B = \frac{1}{2}$, limites para os quais $K_{3,3}$ possui solução. É razoável então conjecturar se para quaisquer valores constantes de δ_A e δ_B, com $\delta_B > \delta_A$, sempre haverá algum grafo bipartido que não é representável. Esta discussão fica em aberto para estudos futuros.

Com o intuito de encontrar uma representação probabilística com complexidade $o(n^2)$ bits, é possível modificar a ideia original para tornar o problema mais fácil.

É possível, por exemplo, ignorar as não-adjacências nos conjuntos independentes do grafo bipartido. Essa informação pode ser codificada como um bit a mais em cada vértice, para informar a qual conjunto independente o vértice faz parte. Assim o problema se resume a representar as adjacências e não-adjacências entre os conjuntos independentes apenas. Com essa modificação, a representação de grafos bipartidos, co-bipartidos e split pode ser exatamente igual, sendo necessária apenas a informação sobre qual tipo de grafo está sendo representado.

De fato, se existir representação probabilística eficiente para grafos bipartidos, é possível transformá-la numa representação para grafos gerais. Esta conclusão segue da transformação apresentada por Ford e Fulkerson em [FF62]. Seja $G' = (V', E')$ a transformação sobre um grafo $G = (V, E)$, tal que para cada vértice $v \in V$, existem $v_1, v_2 \in V'$ e para cada aresta $(u, v) \in E$ existe $(u_1, v_2) \in E'$. É fácil perceber que G' é um grafo bipartido, com $|E'| = 2|E|$ e $|V'| = 2|V|$ (em particular, G' é bipartido completo quando o G é completo). Como existe aresta entre u e v em G se e somente se existe aresta entre u_1 e v_2 em G', então, se existir representação com $O(|V'| \log |V'|)$ bits para G', esta representação também é eficiente em G, pois $O(|V'| \log |V'|) = O(2|V| \log 2|V|) = O(|V| \log |V|)$.

CONCLUSÃO

Estruturas de dados probabilísticas e suas aplicações são um tema bastante popular na indústria atualmente, e embora sua origem remeta à década de 70, os resultados mais significativos foram desenvolvidos a partir dos anos 2000. Por este motivo, ainda há muita teoria a ser debatida sobre este assunto.

É um tema com considerável aplicação prática (o que justifica tamanho interesse da indústria) e com um rico campo teórico a ser explorado, porém ainda carece de abordagem acadêmica mais criteriosa.

Buscamos com este trabalho contribuir com uma aplicação inovadora de estruturas de dados probabilísticas ao problema da representação eficiente de grafos, obtendo resultados importantes, tanto na representação probabilística de grafos gerais quanto de classes específicas.

Contribuições

Ao longo do Capítulo 2, resumimos a literatura disponível de quatro estruturas de dados probabilísticas importantes: filtro de Bloom, *Count-Min sketch*, *MinHash* e *HyperLogLog*. Para cada uma delas, apresentamos a definição, as variantes, o cálculo teórico do erro associado, bem como suas aplicações práticas e resultados experimentais inéditos. Em especial, na Seção 2.4.4, introduzimos uma nova técnica para estimar a cardinalidade da interseção entre conjuntos usando *MinHash* e *HyperLogLog*, com erro relativo apenas ao tamanho da interseção.

Além disso, é introduzida no Capítulo 3 uma nova aplicação para essas estruturas no problema da representação de grafos. Definimos o conceito de *representação implícita probabilística*. Sob esta definição, são desenvolvidas duas representações implícitas probabilísticas. Uma, baseada em filtros de Bloom, é capaz de representar grafos gerais com mesma complexidade da matriz de adjacência no pior caso, sendo ainda mais eficiente para representar grafos esparsos. A outra, baseada em *MinHash*, representa árvores com complexidade inferior à de uma representação implícita, porém possui um fator constante muito grande para ser usada na prática. Demonstramos também que, se existir representação implícita probabilística com complexidade $O(n \log n)$ para grafos bipartidos, co-bipartidos ou split, é possível estendê-la para todas as classes de grafos com a mesma complexidade.

Ao longo da produção deste trabalho, publicamos alguns resultados parciais em congressos. Citamos:

- Resumo *Estruturas de Dados Probabilísticas para Representação de Conjuntos*, aceito

no *I Encontro de Teoria da Computação* e publicado em *Anais do CSBC 2016*.

- Resumo *Estimativa de Cardinalidade da Interseção de Conjuntos Utilizando as Estruturas MinHash e HyperLogLog*, aceito no *XXXVI Congresso Nacional de Matemática Aplicada e Computacional* e publicado em *Anais da SBMAC 2016*.

Trabalhos futuros

Neste trabalho, introduzimos a definição de *representações implícitas probabilísticas*. Sobre este tema, alguns problemas em aberto podem suscitar futuras pesquisas. Podemos citar:

- provar a existência ou não de representação implícita probabilística para grafos bipartidos (na versão estrita do problema) com quaisquer valores constantes de δ_A e δ_B ($\delta_A < \delta_B$);
- buscar novas representações baseadas em *MinHash* para outras classes de grafos além de árvores;
- estudar outras aplicações práticas onde a otimização alcançada pelo uso de filtros de Bloom possa ser útil;
- determinar a viabilidade da versão relaxada de representação implícita probabilística para grafos bipartidos, co-bipartidos ou split, que implica na viabilidade para grafos gerais.

REFERÊNCIAS

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[App12] Austin Appleby. Smhasher and murmurhash, 2012.

[AT08] Christian Antognini and AG Trivadis. Bloom filters. *Internet: http://antognini. ch/papers/BloomFilters20080620. pdf*, 2008.

[AWS15] Inc. Amazon Web Services. Count function documentation, 2015.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[BBT12] DJ Bernstein, Martin Boßlet, and Ruby Core Team. Hash-flooding dos reloaded: attacks and defenses jean-philippe aumasson, kudelski security (nagra), 2012.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM+15] Rémi Bardenet, Odalric-Ambrym Maillard, et al. Concentration inequalities for sampling without replacement. *Bernoulli*, 21(3):1361–1385, 2015.

[BMP+06] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006*, pages 684–695. Springer, 2006.

[Bro97] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[BYJK+02] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.

[CB07] Michele Covell and Sanjeev Baluja. Known-audio detection using waveprint: spectrogram fingerprinting by wavelet hashing. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 1, pages I–237. IEEE, 2007.

[CBWW10] Chih-Yi Chiu, Dimitrios Bountouridis, Ju-Chiang Wang, and Hsin-Min Wang. Background music identification through content filtering and min-hash matching. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 2414–2417. IEEE, 2010.

[CC12] Peter Clifford and Ioana A Cosma. A statistical analysis of probabilistic counting algorithms. *Scandinavian Journal of Statistics*, 39(1):1–14, 2012.

[CCD14] Yousra Chabchoub, Raja Chiky, and Betul Dogan. How can sliding hyperloglog and ewma detect port scan attacks in ip traffic? *EURASIP Journal on Information Security*, 2014(1):1–11, 2014.

[CDF+01] Edith Cohen, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D Ullman, and Cheng Yang. Finding interesting associations without support pruning. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):64–78, 2001.

[CG07] Philippe Chassaing and Lucas Gerin. Efficient estimation of the cardinality of large data sets. *arXiv preprint math/0701347*, 2007.

[CH10] Yousra Chabchoub and Georges Hébrail. Sliding hyperloglog: Estimating cardinality in a data stream, 2010.

[Cha02] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[Cha16] Maurice Chandoo. On the implicit graph conjecture. *arXiv preprint arXiv:1603.01977*, 2016.

[CM03] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252. ACM, 2003.

[CM05] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[CPT11] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. How to apply de bruijn graphs to genome assembly. *Nature biotechnology*, 29(11):987–991, 2011.

[CW77] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112. ACM, 1977.

[DDGR07] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, pages 271–280. ACM, 2007.

[Den13] Frank Denis. Using hyperloglog to detect malware faster than ever, 2013.

[DF03] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *Algorithms-ESA 2003*, pages 605–617. Springer, 2003.

[DGM02] Elias Dahlhaus, Jens Gustedt, and Ross M McConnell. Partially complemented representations of digraphs. *Discrete Mathematics & Theoretical Computer Science*, 5(1):147–168, 2002.

[FCAB98] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 254–265. ACM, 1998.

[FF62] LR Ford and DR Fulkerson. Flows in networks, 1962.

[FFGM08] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 2008.

[FFLS07] Celina Miraglia Herrera de Figueiredo, Guilherme Dias da Fonseca, Manoel José Machado Soares Lemos, and Vinícius Gusmão Pereira de Sá. *Introdução aos Algoritmos Randomizados*. 26º Colóquio Brasileiro de Matemática, 2007.

[FIP12] PUB FIPS. 180-4. *Secure hash standard (SHS)," March*, 2012.

[FKSS99] Wu-chang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. Blue: A new class of active queue management algorithms. *Ann Arbor*, 1001:48105, 1999.

[FM85] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[GIM+99] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[Gir09] Frédéric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.

[GM99] Phillip B Gibbons and Yossi Matias. Synopsis data structures for massive data sets. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 909–910. Society for Industrial and Applied Mathematics, 1999.

[GMP+11] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J Ribeiro, Joshua N Burton, Bruce J Walker, Ted Sharpe, Giles Hall, Terrance P Shea, Sean Sykes, et al. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

[GW95] Michel X Goemans and David P Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6):1115–1145, 1995.

[GWC+10] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. The dynamic bloom filters. *Knowledge and Data Engineering, IEEE Transactions on*, 22(1):120–133, 2010.

[H+96] Nevin Heintze et al. Scalable document fingerprinting. In *1996 USENIX workshop on electronic commerce*, volume 3, 1996.

[HBB+12] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, and Marc Nunkesser. Processing a trillion cells per mouse click. *Proceedings of the VLDB Endowment*, 5(11):1436–1446, 2012.

[Hen06] Monika Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 284–291. ACM, 2006.

[HNH13] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692. ACM, 2013.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[Hon06] Jacob Honoroff. An examination of bloom filters and their applications. *Internet: http://cs. unc. edu/ fabian/courses/CS600. 624/slides/bloomslides. pdf*, 2006.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[Iof10] Sergey Ioffe. Improved consistent sampling, weighted minhash and l1 sketching. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 246–255. IEEE, 2010.

[JB16] Luiz Carlos Irber Junior and C Titus Brown. Efficient cardinality estimation for k-mers in large dna sequencing data sets. *bioRxiv*, page 056846, 2016.

[Jen97] Bob Jenkins. Hash functions. *Dr Dobbs Journal*, 22(9):107–+, 1997.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[KM06a] Adam Kirsch and Michael Mitzenmacher. Distance-sensitive bloom filters. In *ALENEX*, volume 6, pages 41–50. SIAM, 2006.

[KM06b] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[KNR92] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992.

[Knu98] D Knuth. The art of computer programming: Sorting and searching, volume 3, chapter 6.4, 1998.

[KW11] Alexander Klink and Julian Walde. Efficient denial of service attacks on web application platforms. In *The 28th Chaos Communication Congress*, 2011.

[Lem12] Daniel Lemire. The universality of iterated hashing over variable-length strings. *Discrete Applied Mathematics*, 160(4):604–617, 2012.

[LK10] Ping Li and Arnd Christian König. b-bit minwise hashing. In *Nineteenth International World Wide Web Conference (WWW 2010)*. Association for Computing Machinery, Inc., April 2010.

[LKI10] David C Lee, Qifa Ke, and Michael Isard. Partition min-hash for partial duplicate image discovery. In *Computer Vision–ECCV 2010*, pages 648–662. Springer, 2010.

[M+94] Udi Manber et al. Finding similar files in a large file system. In *Usenix Winter*, volume 94, pages 1–10, 1994.

[MAA08] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 618–629. ACM, 2008.

[MGL+10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[Mit02] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

[MJDS07] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pages 141–150. ACM, 2007.

[Mor78] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[MP80] J Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[MS10] Kathy Macropol and Ambuj Singh. Scalable discovery of best clusters on large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):693–702, 2010.

[Mul88] John Harold Muller. Local structure in graph classes, 1988.

[Mul93] James K Mullin. Estimating the size of a relational join. *Information Systems*, 18(3):189–196, 1993.

[OTM+16] Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome Biology*, 17(1):132, 2016.

[Pas13] Andrew Pascoe. Hyperloglog and minhash, 2013.

[PHCK+12] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[PSN10] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *Distributed and Parallel Databases*, 28(2-3):119–156, 2010.

[PTW01] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[R+13] Alexandre José Monteiro Rodrigues et al. Recomendação de conteúdos: aplicação de agrupamento distribuído a conteúdos de tv, 2013.

[Ram89] MV Ramakrishna. Practical performance of bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237–1239, 1989.

[RD07] Florin Rusu and Alin Dobra. Statistical analysis of sketch estimators. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 187–198. ACM, 2007.

[RUUU12] Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of massive datasets*, volume 77. Cambridge University Press Cambridge, 2012.

[RV96] Raimundo Real and Juan M Vargas. The probabilistic basis of jaccard's index of similarity. *Systematic biology*, pages 380–385, 1996.

[RW98] Alex Rousskov and Duane Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22):2155–2168, 1998.

[SH15] CS Sindhu and Nagaratna P Hegde. A brief insight into computational tools in big data, 2015.

[SHM] Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. Popularity is everything.

[SKP15] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full sha-1. Cryptology ePrint Archive, Report 2015/967, 2015.

[SL14] Anshumali Shrivastava and Ping Li. In defense of minhash over simhash. *arXiv preprint arXiv:1407.4416*, 2014.

[Spi03] Jeremy P Spinrad. *Efficient graph representations*. American mathematical society, 2003.

[SVG08] Osman Salem, Sandrine Vaton, and Annie Gravey. A novel approach for anomaly detection over high-speed networks. In *Proceedings of the 3rd European Conference on Computer Network Defense*, pages 49–68. Springer, 2008.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases, 1992.

[TSMJ12] Carlos HC Teixeira, Arlei Silva, and Wagner Meira Jr. Min-hash fingerprints for graph kernels: A trade-off among accuracy, efficiency, and compression. *Journal of Information and Data Management*, 3(3):227, 2012.

[WVZT90] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 19–35. Springer, 2005.

[WZL13] Xin-Jing Wang, Lei Zhang, and Ce Liu. Duplicate discovery on 2 billion internet images. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2013 IEEE Conference on*, pages 429–436. IEEE, 2013.

[ZOWX06] Qi George Zhao, Mitsunori Ogihara, Haixun Wang, and Jun Jim Xu. Finding global icebergs over distributed data sets. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 298–307. ACM, 2006.

[ZPCK+14] Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C Titus Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS one*, 9(7):e101271, 2014.