UNIVERSIDADE FEDERAL DO PARANÁ

LEANDRO MIRANDA ZATESKO

ESQUEMAS DE *HASHING* PERFEITOS, MÍNIMOS, PRÁTICOS, DETERMINÍSTICOS E EFICIENTES EM TEMPO E EM ESPAÇO

CURITIBA

2011

UNIVERSIDADE FEDERAL DO PARANÁ

LEANDRO MIRANDA ZATESKO

ESQUEMAS DE *HASHING* PERFEITOS, MÍNIMOS, PRÁTICOS, DETERMINÍSTICOS E EFICIENTES EM TEMPO E EM ESPAÇO

Dissertação apresentada ao Curso de Pós-Graduação em Informática, Área de Concentração em Inteligência Computacional, Departamento de Informática, Setor de Ciências Exatas, Universidade Federal do Paraná, como parte das exigências para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Jair Donadelli Jr.

CURITIBA

2011

Z38

Zatesko, Leandro Miranda
Esquemas de *hashing* perfeitos, mínimos, práticos, determinísticos e eficientes em tempo e em espaço / Leandro Miranda Zatesko. –
Curitiba: 2011.
87 f.
Orientação Dr. Jair Donadelli Jr.
Dissertação (Mestrado em Informática) - Setor de Tecnologia da Universidade Federal do Paraná, 2011.

Inclui bibliografia.

1.Hash. 2. Algorítimos. I. Título.

CDD 511.8

Catalogação na fonte pela Biblioteca de Ciência e Tecnologia da UFPR

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Leandro Miranda Zatesko, avaliamos o trabalho intitulado, *"ESQUEMAS DE HASHING PERFEITOS, MINÍMOS, PRÁTICOS, DETERMINÍSTICOS E EFICIENTES EM TEMPO E EM ESPAÇO"*, cuja defesa foi realizada no dia 17 de novembro de 2011, às 14:00 horas, no Departamento de informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 17 de novembro de 2011

Prof. Dr. Jair Donadelli
DINF/UFPR – Orientador

Prof. Dr. Murilo Vicente Gonçalves da Silva
Dainf/UTFPR– Membro Externo

Prof. Dr. André Luiz Pires Guedes
DINF/UFPR

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

TERMO DE APROVAÇÃO

LEANDRO MIRANDA ZATESKO

ESQUEMAS DE HASHING PERFEITOS, MÍNIMOS, PRÁTICOS, DETERMINÍSTICOS E EFICIENTES EM TEMPO E EM ESPAÇO

Dissertação aprovada como parte das exigências para a obtenção do título de Mestre no Curso de Pós-Graduação em Informática, Setor de Ciências Exatas da Universidade Federal do Paraná, pela seguinte banca examinadora:

Orientador: Prof. Dr. Jair Donadelli Jr.
Centro de Matemática, Computação e Cognição, UFABC

Prof. Dr. André Luiz Pires Guedes
Departamento de Informática, UFPR

Prof. Dr. Murilo Vicente Gonçalves da Silva
Departamento Acadêmico de Informática, UTFPR

Curitiba, 17 de novembro de 2011.

A Deus,
pois Dele, por Ele e
para Ele são todas as coisas.

AGRADECIMENTOS

Aos meus queridos pais, Luciano e Vera Zatesko, por tanto me abençoarem com amor, apoio, sustento e educação.

À minha noiva, Giullia, por me amar e me aceitar incondicionalmente.

Ao professor Dr. Jair Donadelli Jr., por todos os anos de orientação acadêmica e por toda a confiança em mim depositada.

Ao professor Dr. Fabiano C. Botelho, não apenas pelos trabalhos que fortemente inspiraram este meu, mas também pela preciosíssima ajuda que me concedeu ao responder minhas correspondências.

Aos professores do Grupo de Pesquisa em Algoritmos (ARG), Dr. André Luiz P. Guedes, Dr. André Luís Vignatti e Dr. Renato Carmo, pelo auxílio que tenho recebido nesses anos todos, e aos demais professores do Departamento de Informática da UFPR (DInf), por todo o conhecimento que gentilmente já comigo compartilharam.

À minha família: meus irmãos, Luciana e Leônidas, e ao meu cunhado, Alex, por trilharem comigo minha jornada; minha estimadíssima tia, Cleide, por tudo o que me ensinou e por toda a devoção que me dedicou até aqui; meus padrinhos, Donizete e Catarina Maia, por sempre acreditarem em mim; meus demais familiares, por todo o apreço que têm por mim.

À família Picco, pelo carinho e pelo suporte que incontáveis vezes me levantaram, e aos demais irmãos que tenho em Jesus Cristo, pelas sinceras orações.

Aos meus prezados companheiros de laboratório, especialmente Sílvio, Francieli, Bruna, Márcia e Maysa, pelos conselhos, pela amizade e pela torcida por este trabalho.

A Jucélia Miecznikowski, secretária do Programa de Pós-Graduação em Informática, pelo caro trabalho que me manteve no programa, e aos demais funcionários do DInf, pelo zelo com que mantêm a infra-estrutura e os laboratórios.

À CAPES, pelo auxílio financeiro.

Se te parece que sabes e entendes bem muitas coisas,
lembra-te que é muito mais o que ignoras.

Imitação de Cristo

RESUMO

Este trabalho propõe algoritmos determinísticos que, dado um conjunto com n chaves, constroem em tempo esperado $O(n)$ uma função *hash* com tempo de busca no pior caso $O(1)$, a qual mapeia sem colisão as chaves para o conjunto $\{0, \ldots, n-1\}$. Esses esquemas de *hashing* perfeitos e mínimos são meras variantes dos esquemas aleatorizados de Botelho, Kohayakawa e Ziviani (2005) e Botelho, Pagh e Ziviani (2007) e mostraram resultados empíricos equivalentes aos dos algoritmos originais. As variantes determinísticas foram implementadas a partir dos códigos dos esquemas originais desenvolvidos na biblioteca CMPH pelos próprios autores, a qual é mantida no `SourceForge.net`. Todos os esquemas foram alimentados com os mesmos conjuntos de chaves, para que pudessem ser comparados com justiça. Foram executados testes para conjuntos com até $25\,000\,000$ de chaves. Ademais, os esquemas propostos contam evidentemente com a vantagem de sempre produzirem a mesma *hash* para um mesmo conjunto de chaves. Esse comportamento determinístico pode ser útil para o desenvolvimento dum esquema dinâmico de *hashing*, em que figuram operações como inserção e deleção de chaves, inspirado num dos excelentes esquemas estáticos abordados. Um dos esquemas de Botelho, Pagh e Ziviani (2007), por exemplo de excelência, constrói *hashes* representáveis por apenas aproximadamente $2{,}62$ *bits* por chave. Tal resultado é muito próximo da cota inferior justa conhecida, de aproximadamente $1{,}44$ *bits* por chave. Tanto as versões determinísticas propostas quanto as originais mostram-se práticas para aplicações reais de *Hashing*. No entanto, na fundamentação teórica do trabalho de Botelho, Kohayakawa e Ziviani (2005) ainda restava uma conjectura. A presente dissertação também propõe uma demonstração para a conjectura e encerra a corretude do esquema.

Palavras-chave: *Hashing*. Desaleatorização. Análise de algoritmos.

ABSTRACT

This work proposes deterministic algorithms that, given a set with n keys, build in expected time $O(n)$ a hash function with worst-case $O(1)$ lookup time, which maps without collision the keys to the set $\{0, \ldots, n-1\}$. These minimal perfect hashing schemes are mere variants of the probabilistic schemes of Botelho, Kohayakawa e Ziviani (2005) and Botelho, Pagh e Ziviani (2007) and have showed empirical results equivalent to the original algorithms. The deterministic variants have been implemented from the source codes of the original schemes developed in CMPH library by the authors theirselves, which is maintained at `SourceForge.net`. All the schemes have been input with the same key sets, so they could be compared fairly. Tests have been executed over datesets with up to $25\,000\,000$ keys. Moreover, the proposed schemes have evidently the advantage of always generating the same hash function for the same key set. This deterministic behaviour might be useful for developing a dynamic hashing scheme, in which lie operations like insertion and deletion of keys, inspired in one of the excelent static schemes covered. One of the schemes of Botelho, Pagh e Ziviani (2007), as an example of excellence, builds hash functions representable by only about 2.62 bits for key. Such a result is very close to the known tight lower bound, of about 1.44 bits for key. Both the deterministic proposed versions and the original ones show to be practical for common applications of Hashing. However, in the theoretical foundation of the work of Botelho, Kohayakawa e Ziviani (2005) a conjecture remained open. The present dissertation also proposes a proof for the conjecture and ends up the correctness of the scheme.

Keywords: Hashing. Derandomization. Algorithms analysis.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

URL — *Uniform Resource Locator* (*veja* p. 28)

FKS — esquema proposto por Fredman, Komlòs e Szemerédi (1984) (*veja* p. 38)

BMZ — esquema proposto por Botelho, Kohayakawa e Ziviani (2005) (*veja* p. 48)

CMPH — biblioteca de esquemas mínimos e perfeitos escritos em C (*veja* p. 49)

BKZ — sinônimo de BMZ (*veja* p. 49)

MOS — abordagem *Mapping, Ordering and Searching* (*veja* p. 49)

J-BMZ — esquema BMZ com as *hashes* de Jenkins para o mapeamento (*veja* p. 79)

BDZ — esquema proposto por Botelho, Pagh e Ziviani (2007) (*veja* p. 80)

r-BDZ — esquema da família do BDZ que utiliza r-hipergrafos (*veja* p. 80)

BPZ — sinônimo de BDZ (*veja* p. 80)

J-BDZ — esquema BDZ com as *hashes* de Jenkins para o mapeamento (*veja* p. 106)

UFPR — Universidade Federal do Paraná (*veja* p. 107)

D-BMZ — versão determinística do BMZ (*veja* p. 115)

D-BDZ — versão determinística do BDZ (*veja* p. 120)

D-J-BMZ — versão determinística do J-BMZ (*veja* p. 123)

D-J-BDZ — versão determinística do J-BDZ (*veja* p. 123)

TLD — *Top-Level Domain* (*veja* p. 124)

RAM — *Random Access Machine* (*veja* p. 149)

LISTA DE SÍMBOLOS

U — conjunto universo (*veja* p. 24)

M — tabela *hash* (*veja* p. 24)

$h\colon U \to M$ — função *hash* (*veja* p. 24)

m — tamanho da tabela *hash* M (*veja* p. 24)

u — tamanho do universo U (*veja* p. 24)

n — quantidade de chaves (*veja* p. 24)

S — conjunto de chaves (*veja* p. 24)

$x \in S$ — chave (*veja* p. 24)

$h(x)$ — endereço associado à chave x pela *hash* h (*veja* p. 24)

Σ^* — conjunto das palavras formadas pelos elementos de Σ (*veja* p. 25)

L — tamanho máximo de um a qualquer em U (*veja* p. 25)

Σ_L^* — conjunto das palavras sobre Σ com tamanho no máximo L (*veja* p. 25)

$h(S)$ — imagem de S por h (*veja* p. 26)

$h^{-1}(i)$ — imagem inversa de i por h (*veja* p. 27)

S_i — bucket de colisão de i em relação a S (*veja* p. 27)

$\lg x$ — logaritmo de x na base 2 (*veja* p. 35)

G_{crit} — subgrafo crítico de G (*veja* p. 49)

G_{ncrit} — subgrafo não-crítico de G (*veja* p. 49)

E_{crit} — conjunto das arestas críticas (*veja* p. 49)

E_{ncrit} — conjunto das arestas não-críticas (*veja* p. 49)

V_{scrit} — $V(G_{\text{crit}}) \cap V(G_{\text{ncrit}})$ (*veja* p. 49)

V_{crit} — conjunto dos vértices críticos (*veja* p. 49)

V_{ncrit} — conjunto dos vértices não-críticos (*veja* p. 49)

$E(G)$ — conjunto de arestas de G (*veja* p. 50)

$\mathscr{G}(V, n)$ — todos os grafos com n arestas e conjunto de vértices V (*veja* p. 51)

c — no BMZ, $|V(G)| = cn$, sendo $n = |S| = |E(G)|$ e $c = 1{,}15$ (*veja* p. 54)

$G - v$ — remoção dum vértice v do grafo G (*veja* p. 56)

$g\colon V(G) \to \mathbb{Z}$ — rotulação dos vértices no BMZ (*veja* p. 59)

$\boldsymbol{A}$ — valores que a parte crítica da busca do BMZ não usa (*veja* p. 63)

A — conjunto dos valores em $\boldsymbol{A}$ (*veja* p. 64)

A_E — valores que a parte crítica da busca do BMZ usa para h (*veja* p. 66)

$I(u)$ — número de reassociações para rotular u no BMZ (*veja* p. 69)

N_{t} — soma total de reassociações feitas pelo BMZ (*veja* p. 69)

N_{bedges} — número de arestas de retorno na fase crítica do BMZ (*veja* p. 69)

e — função da fase de mapeamento do r-BDZ (*veja* p. 81)

$c(r)$ — no r-BDZ, $|V(G_r)| = c(r)n$, dada a Equação 2.93 (p. 87) (*veja* p. 81)

L — ordenação das arestas no mapeamento do r-BDZ (*veja* p. 81)

ρ — função da fase de designação do r-BDZ (*veja* p. 81)

g — rotulação dos vértices no r-BDZ (*veja* p. 82)

rank — função da fase de *ranking* do r-BDZ (*veja* p. 82)

$\mathbb{E}I_{\mathsf{BDZ\text{-}map}}(r)$ — esperança de iterações no mapeamento do BDZ (*veja* p. 87)

$h'(x)$ — cadeia de *bits* de onde se tiram $h_0, \ldots, h_{r-1}$ (*veja* p. 101)

γ — no r-BDZ, a constante que define o tamanho de $h'(x)$ (*veja* p. 101)

$\binom{A}{n}$ — conjunto dos subconjuntos de A de tamanho n (*veja* p. 135)

$K\,[i]$ — i-ésimo *byte* da cadeia de *bytes* K (*veja* p. 146)

$K \gg d$ — deslocamento de K d *bits* para a direita (*veja* p. 146)

w_1+w_2 — soma com *overflow* das palavras w_1 e w_2 (*veja* p. 146)

w_1-w_2 — diferença com *overflow* das palavras w_1 e w_2 (*veja* p. 146)

$w_1\hat{}w_2$ — *xor bit* a *bit* de w_1 com w_2 (*veja* p. 146)

w+(b_1, b_2, b_3, b_4) — w+ a concatenação de $b_1, \ldots, b_4$ (*veja* p. 147)

φ — razão áurea (*veja* p. 147)

$a \ll d$ — deslocamento de a d *bits* para a esquerda (*veja* p. 148)

J_1, J_2, J_3 — *hashes* de Jenkins (*veja* p. 148)

2^A — conjunto dos subconjuntos de A (*veja* p. 158)

G_r — um hipergrafo em que toda aresta possui r vértices (*veja* p. 159)

SUMÁRIO

1 INTRODUÇÃO

Um *grafo* é uma estrutura matemática composta por vértices e arestas, sendo cada aresta uma ligação entre dois e somente dois vértices distintos. A Teoria dos Grafos pode ser uma poderosa ferramenta para a Ciência da Computação em situações nas quais objetos duma certa coleção estabeleçam alguma relação *dois-a-dois* entre si. Por exemplo, podemos verificar a ocorrência de *deadlocks* em sistemas operacionais modelando as requisições e as alocações de recursos por processos através de um grafo, conhecido como *grafo de alocação de recursos*, em que uma aresta dirigida — ou direcionada — de um processo P a um recurso R significa uma requisição e uma aresta dirigida de um recurso a um processo significa uma alocação. Por exemplo, Silberschatz e Galvin (2000), mostram que, quando há ciclos nesse grafo, como destacado na Figura 1.1, um processo *pode* esperar eternamente um recurso ser liberado por outro processo, configurando um *deadlock*.

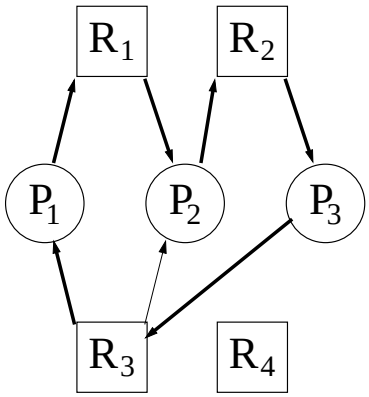


FIGURA 1.1 — Um *deadlock* constatado num grafo de alocação de recursos.

Como não trabalharemos com arestas dirigidas, podemos considerar que uma aresta é um conjunto de dois vértices. Um modo de se representar computacionalmente um grafo é através de sua matriz de adjacências, uma matriz em que a posição $a_{ij} = 1$ se a aresta $\{i, j\}$ existe no grafo ou $a_{ij} = 0$ caso contrário. Por exemplo, a matriz

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{1.2}$$

é a matriz de adjacências do grafo representado pela Figura 1.3, cujos vértices são 1, 2, 3, 4, 5 e 6 e cujas arestas são $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, $\{4, 1\}$, $\{1, 3\}$ e $\{5, 6\}$.

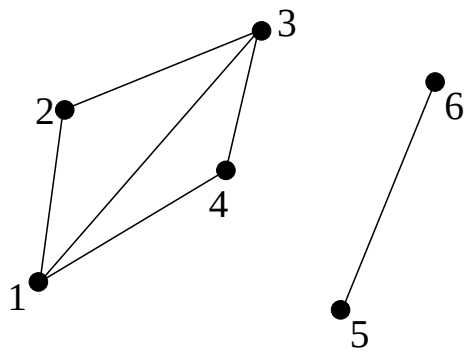


FIGURA 1.3 — Um exemplo de grafo.

Pudemos construir a matriz de adjacências da Equação 1.2 porque o conjunto dos vértices do grafo, $\{1, \ldots, 6\}$, coincidiu com o conjunto dos índices de uma matriz 6×6. Se o conjunto de vértices fosse, por exemplo, $\{2, 4, 6, 8, 10, 12\}$, precisaríamos construir um mapeamento um-para-um de $\{2, 4, 6, 8, 10, 12\}$ em $\{1, 2, 3, 4, 5, 6\}$ que fornecesse os índices de cada vértice na matriz. Nesse caso, seria fácil: o mapeamento poderia ser a função $h\colon x \to \frac{x}{2}$. Porém, o conjunto de vértices pode ser

qualquer conjunto finito, como, por exemplo, $\{\texttt{john}, \texttt{wins}, \texttt{ton}, \texttt{ono}, \texttt{len}, \texttt{non}\}$. Construir mapeamentos como esses de modo eficiente é um dos objetivos do estudo de *Hashing*.

Uma *hash* é um mapeamento de um universo de elementos para uma tabela finita, chamada de *tabela hash*. Matematicamente, tanto o universo de elementos quanto a tabela *hash* podem ser vistos como conjuntos. Assim, uma *hash* nada mais é que uma função cujo domínio é o conjunto universo e cujo contradomínio é a tabela *hash*. Por exemplo, embora a relação ilustrada na Figura 1.4(a) possa ser considerada uma *hash*, as relações ilustradas nas Figuras 1.4(b) e 1.4(c) não podem nem mesmo ser consideradas funções.

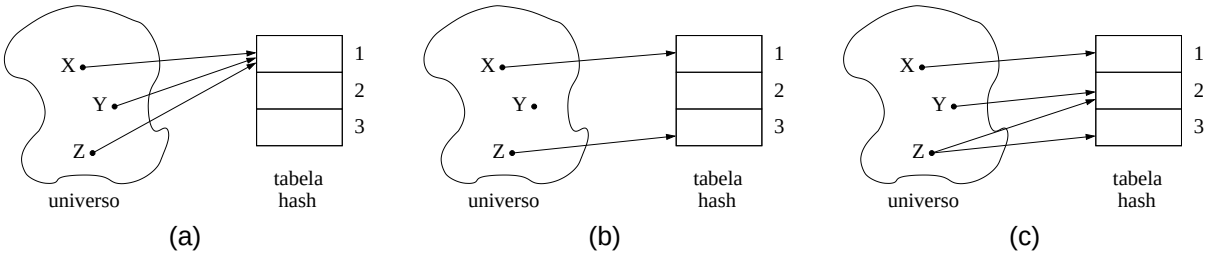


FIGURA 1.4 — Exemplos de relações entre um universo e uma tabela.

A palavra inglesa *hash* [hˈæːʃ] deriva da francesa *hache*, que significa *machado*. O verbo *to hash*, sinônimo de *to chop*, pode ser traduzido como *tornar em pequenos pedaços* ou *picar*. Em nosso contexto, usamos esta palavra porque, fazendo *hashing*, geralmente tornamos um elemento *grande* em um *pedaço pequeno* que o designa. Um bom exemplo é a *hash* que mapeia seres humanos a suas impressões digitais. Armazenar computacionalmente um ser humano é difícil, mas armazenar uma impressão digital é muito mais fácil. A propósito, *hashing* é o nome que se dá tanto ao processo quanto ao problema de se encontrar uma *hash*.

É comum encontrarmos na Literatura em língua portuguesa os termos *função de espalhamento* e *tabela de espalhamento*, como traduções para *hash* e *hash table* (tabela *hash*), respectivamente. No entanto, como a comunidade científica brasileira já

consagrou os termos em inglês, neste trabalho não usaremos as traduções existentes.

Embora uma *hash* seja definida sobre todo um conjunto universo, estamos geralmente interessados no comportamento duma *hash* sobre um conjunto específico de elementos, chamados de *chaves*. Por exemplo, a *hash* $h\colon x \to \frac{x}{2}$, definível sobre todo o universo dos reais, apresenta um bom comportamento para o caso específico em que o conjunto de chaves é $V(G) = \{2, 4, 6, \ldots, 2|V(G)|\}$.

Consideremos agora a problemática dum professor que, para dificultar as colas nas provas aplicadas a uma turma muito grande, elaborou 26 diferentes modelos de prova. Ele deseja atribuir a cada aluno da turma um número de 1 a 26, que corresponda ao modelo de prova que o aluno terá de resolver.

Uma maneira de mapear esses alunos à tabela $\{1, \ldots, 26\}$ é seguir a ordenação natural das 26 letras do alfabeto. Assim, como ilustrado na Figura 1.5, recebe o número i o aluno cujo nome começar com a i-ésima letra do alfabeto, a despeito de diacríticos — se houver alguma aluna de nome Ártemis, receberá o número 1. Note-se que, embora o professor esteja interessado apenas nos mapeamentos sobre os nomes dos alunos de sua turma, o mapeamento estabelecido funciona para qualquer nome escrito em nosso alfabeto.

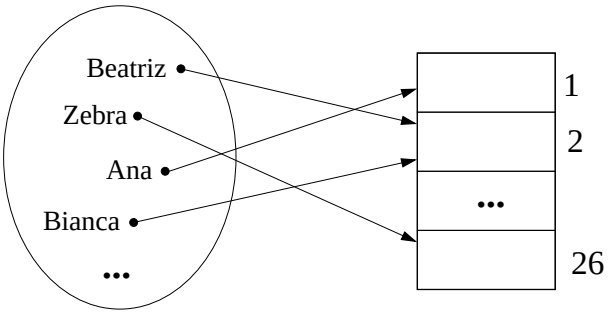


FIGURA 1.5 — Uma *hash* que mapeia nomes de alunos a números.

No exemplo que descrevemos, tivemos por *hash* o mapeamento que leva um nome ao ordinal de sua primeira letra no alfabeto; por universo, a classe — infinita —

de todos os nomes possíveis formados com as letras de nosso alfabeto; por tabela *hash*, o conjunto $\{1, \ldots, 26\}$; por chave, cada nome de aluno da específica turma em questão.

Deparamo-nos cotidianamente com situações que envolvem *Hashing*. A maioria das bibliotecas cataloga seus livros utilizando um procedimento que leva em conta várias entradas, como a seção na qual o livro foi classificado, o sobrenome do autor, o título do livro e o número do exemplar. Esse procedimento, definido para ser aplicável a qualquer livro, é também mais um exemplo de *hash*.

1.1 ALGUMAS DEFINIÇÕES EM TEORIA DE *HASHING*

A seguir apresentamos a definição formal de *hash*.

DEFINIÇÃO 1.6 (*Hash*): Sendo U um conjunto, chamado de universo, e M um conjunto finito, chamado de tabela *hash*, uma *hash* é uma função $h\colon U \to M$.

Como a tabela *hash* é por definição finita, assumimos sem perda de generalidade que $M = \{0, \ldots, m-1\}$, sendo m o tamanho da tabela. U, no entanto, embora possa ser infinito, na maioria das aplicações é restrito para um conjunto finito de tamanho u. O interesse de nosso estudo, contudo, reside no comportamento de uma função *hash* para com subconjuntos de um predeterminado tamanho n. Considerando um subconjunto arbitrário S de U, chamamos cada $x \in S$ de *chave*, assim como dizemos que cada $h(x)$ é o *endereço* associado à chave x pela *hash* h.

No presente trabalho, à exceção da Seção 2.1 (p. 38), convencionamos que cada elemento do conjunto universo é uma palavra de tamanho finito não-nulo formada

por caracteres dum alfabeto também finito Σ. Usamos Σ^* para denotar o conjunto de todas as palavras formadas por caracteres de Σ. Também limitamos o tamanho de cada palavra a uma constante L. Assim, nosso universo U sempre será o conjunto Σ_L^* de todas as palavras formadas por no máximo L caracteres de Σ.

Limitando os tamanhos das chaves por uma constante L, podemos dizer que operações como ler uma chave ou um elemento qualquer do universo custam tempo assintoticamente constante. Aos leitores não familiarizados com Análise de Complexidade de Tempo de Algoritmos, recomendamos a leitura do Apêndice C e dos livros de Papadimitriou (1995, cap. 1), de Cormen, Leiserson e Rivest (1990, cap. 2). Àqueles não familiarizados com Notação Assintótica, por sua vez, recomendamos o livro de Graham, Knuth e Patashnik (1994, cap. 9) e também o Apêndice C.

Na Seção 2.1, consideraremos os elementos do universo não como palavras, mas como números naturais. Há vários modos pelos quais podemos entender uma palavra de Σ_L^* como um natural. Por exemplo, se enumerarmos Σ de 0 até $|\Sigma| - 1$ e trocarmos os caracteres das palavras pelos respectivos números decorrentes da enumeração, poderemos interpretar cada palavra como sendo um número natural escrito na base $|\Sigma|$. Na referida seção, também tomaremos $U = \{0, \ldots, u-1\}$. O Quadro 1.7 ilustra como U pode ser considerado tanto Σ_L^* quanto $\{0, \ldots, u-1\}$ sem perda alguma de generalidade.

$\{\mathtt{e}, \mathtt{m}, \mathtt{a}, \mathtt{c}, \mathtt{s}\}_3^*$	$\mathtt{e} \mapsto 0$, $\mathtt{m} \mapsto 1$, $\mathtt{a} \mapsto 2$, $\mathtt{c} \mapsto 3$, $\mathtt{s} \mapsto 4$	$\{0, \ldots, 124\}$
`mac`	$(123)_5$	38
`sms`	$(414)_5$	109
`cce`	$(330)_5$	90

QUADRO 1.7 — Como mapear $\{e, m, a, c, s\}_3^*$ para $\{0, \ldots, 124\}$.

Uma *busca* — em inglês *query* ou *lookup* — é o procedimento computacional que, dada uma chave x, computa o endereço $h(x)$. Note-se que o uso da palavra

busca é um tanto que infeliz, já que não necessariamente o endereço computado $h(x)$ designa x, pois pode haver uma outra chave y tal que $h(x) = h(y)$, como trataremos adiante. Também não se deve confundir o termo *busca*, nesse contexto, com a fase de busca dos esquemas que utilizam a abordagem MOS, apresentados na Secão 2.2 (p. 49). Em inglês essa confusão não ocorre, já que a terceira fase dum esquema MOS chamamos de *searching*.

No estudo de *Hashing* também importa a complexidade computacional das buscas. Como assumimos um limitante superior L para os tamanhos das chaves, desejamos que as buscas tenham complexidade computacional $O(1)$. Afinal, já que ler uma chave x pode ser feito em tempo assintoticamente constante, queremos também que computar o endereço $h(x)$ possa ser feito em tempo assintoticamente constante.

1.1.1 Casos particulares de *hash*

Notemos, no exemplo da Figura 1.5 (p. 23), que as chaves $\mathrm{Beatriz}$ e Bianca são ambas mapeadas para o número 2 da tabela *hash*. Esse fenômeno, de duas chaves distintas serem mapeadas para o mesmo endereço, chamamos de *colisão*, assim como chamamos as chaves envolvidas de *sinônimas*. Uma *hash* é dita *perfeita* sobre um conjunto de chaves S quando $h(x) \neq h(y)$, para todo par de chaves $x, y \in S$ tais que $x \neq y$.

Usamos $h(S)$ para denotar a imagem de S por h:

$$h(S) = \{h(x) : x \in S\}. \tag{1.8}$$

Uma *hash* é dita *mínima* sobre um conjunto de chaves S quando não sobram ele-

mentos na tabela M que não sejam associados a alguma chave de S pela *hash* h — noutras palavras, quando $h(S) = M$.

Dado um $i \in M$, usamos $h^{-1}(i)$ para denotar a imagem inversa de i por h: o conjunto $h^{-1}(i) = \{a \in U : h(a) = i\}$. Note-se que as imagens inversas dos elementos da tabela *hash* particionam o conjunto universo. Na Figura 1.9, $h^{-1}(1) = \{T, E, X\}$, $h^{-1}(2) = \{C, L, S\}$ e $h^{-1}(3) = \{I, N, D\}$.

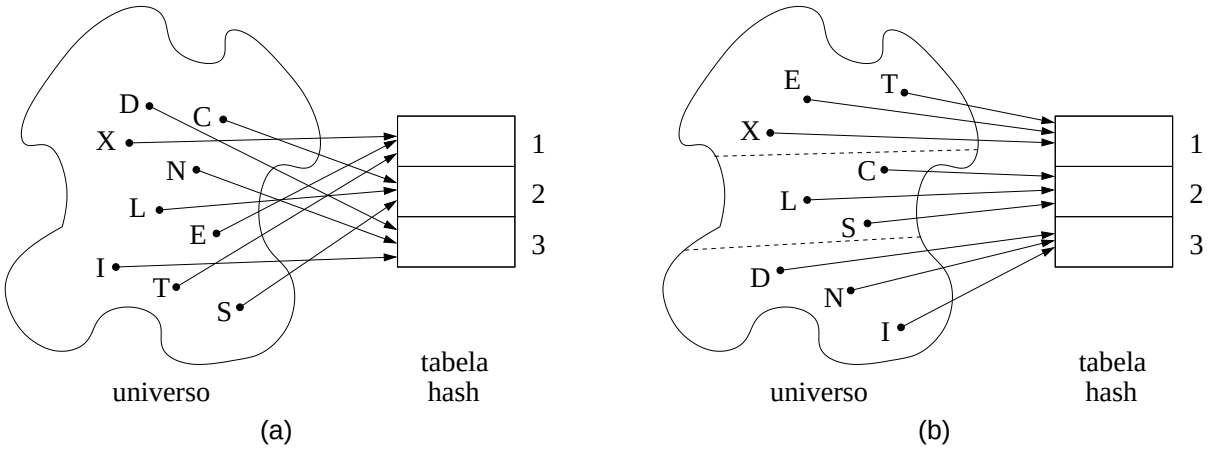


FIGURA 1.9 — Como uma *hash* particiona um conjunto universo.

DEFINIÇÃO 1.10 (*Bucket* de colisão): Dado um $i \in M$ e um $S \subseteq U$, o *bucket de colisão* da posição i em relação a S, denotado por S_i, é o conjunto

$$S_i = \{x \in S : h(x) = i\}. \tag{1.11}$$

Alternativamente,

$$S_i = h^{-1}(i) \cap S. \tag{1.12}$$

No exemplo da Figura 1.9, se tomamos $S = \{S, E, X\}$, temos

$$S_1 = \{E, X\}, \qquad S_2 = \{S\}, \qquad \text{e} \qquad S_3 = \emptyset. \tag{1.13}$$

Equivalentemente, podemos definir que uma *hash* $h\colon U \to M$ é, sobre um conjunto de chaves $S \subset U$,

a) perfeita, quando h opera injetivamente sobre S;

b) mínima, quando h opera sobrejetivamente sobre S.

Dessarte, se h é perfeita sobre S, então, $n \leqslant m$, da mesma forma que, se é mínima, então, $n \geqslant m$. Assim, h só pode ser ao mesmo tempo perfeita e mínima — e, portanto, bijetiva — sobre S se $n = m$.

Suponhamos que queremos armazenar 25 milhões de *URLs* — *URL*: *Uniform Resource Locator* — num vetor $\boldsymbol{v}$. Uma maneira trivial de se fazer isso é armazenar a primeira *URL* lida em $\boldsymbol{v}[0]$, a segunda *URL* em $\boldsymbol{v}[1]$ e assim sucessivamente (Algoritmo 1.14). Esse procedimento parece bom, pois pode ser executado em tempo linear na quantidade de *URLs*, sem contar que não desperdiça posições de $\boldsymbol{v}$, que não precisa ter mais que 25 milhões de posições. No entanto, se, depois de armazenadas as *URLs*, quisermos encontrar uma *URL* específica, precisaremos percorrer as posições de $\boldsymbol{v}$ uma a uma. No pior caso, precisaremos percorrer todas as 25 milhões de posições.

ALGORITMO 1.14 — Um algoritmo que armazena *URLs* num vetor.

ENTRADA: Um conjunto S com 25 milhões de *URLs* e um vetor $\boldsymbol{v}$ de 25 milhões de posições vazio.
SAÍDA: O vetor $\boldsymbol{v}$ preenchido com as *URLs*.

1: $j \leftarrow 0$.
2: ENQUANTO $S \neq \emptyset$, FAÇA:
3: leia a próxima *URL* x de S;
4: armazene x em $\boldsymbol{v}[j]$;
5: $S \leftarrow S \setminus \{x\}$;
6: $j \leftarrow j + 1$,
7: FIM.

Em problemas como esse, a abordagem de *Hashing* torna-se bastante atraente. O desejado é que encontremos uma *hash* eficientemente computável que mapeie uma *URL* num número, o qual poderá ser interpretado como o índice da *URL* no vetor.

Note-se que a definição da função *hash* precisa valer para qualquer *URL* de tamanho no máximo uma constante L, e não apenas para as 25 milhões da entrada — nosso conjunto de chaves. Porém, olharemos apenas para os endereços associados às chaves, pois apenas elas armazenaremos no vetor.

Tomemos Σ o conjunto dos possíveis caracteres que formam uma *URL* (letras do nosso alfabeto e caracteres especiais, como $:$, $/$, $.$, $?$ etc.). Se estabelecemos uma ordenação sobre Σ, contando os elementos de Σ de 0 até $|\Sigma| - 1$, podemos entender cada caractere de uma *URL* como um dígito e cada *URL* como um número na base $|\Sigma|$. Desse modo, temos nossa *hash*. Ela é perfeita e também muito eficientemente computável. Cada busca pode ser feita em tempo linear no tamanho da *URL* buscada, tempo que pode ser considerado constante, já que limitamos os tamanhos das chaves por L.

Todavia, a tabela *hash* precisa ser grande o suficiente para conter o maior endereço associado a uma *URL*: $|\Sigma|^L - 1$. Se Σ fosse composto apenas pelas letras minúsculas do alfabeto, e se o tamanho de uma *URL* não ultrapassasse 10 caracteres, esse número seria $26^{10} - 1 = 141\,167\,095\,653\,375$, muitíssimo maior que $25\,000\,000$. Nossa *hash*, portanto, está muito longe de ser mínima, e, se ainda a queremos utilizar para guardar as *URLs* num vetor, a abordagem proposta se torna inviável.

1.2 ESQUEMAS DE *HASHING*

Mostramos acima um exemplo duma *hash* que é perfeita para todo conjunto de chaves S, mas que fica muito longe de ser mínima. Se queremos uma *hash* que seja perfeita para muitos conjuntos de chaves, precisamos que m seja muito maior que n. Por outro lado, se queremos que seja mínima para muitos conjuntos de chaves,

precisamos que m seja muito menor que n. Esse problema se deve ao fato de que temos tratado de construir as *hashes* sobre U independentemente dos conjuntos S. Essa abordagem, no entanto, faz com que, na maioria dos casos reais de aplicação de *Hashing*, tenhamos tanto uma frequência considerável de colisões quanto um desperdício considerável de espaço na tabela M.

Uma abordagem alternativa é não determinar *hashes* que sirvam para todos os conjuntos de chaves, mas estipular um *esquema de hashing* — também chamado de *método de hashing* — que produza, em tempo aceitável, uma *hash* para cada conjunto S. Assim, poderemos inclusive construir *hashes* perfeitas e mínimas. Quando um esquema, para todo conjunto S, sempre gera uma *hash* perfeita, dizemos que o esquema é perfeito. De igual modo, quando as *hashes* geradas são sempre mínimas, o esquema é dito mínimo.

DEFINIÇÃO 1.15 (Esquema de *hashing*): Um esquema de *hashing* é um algoritmo que, recebendo como entrada um universo U, uma tabela M e um conjunto de chaves S, fornece como saída uma *hash* $h\colon U \to M$ com as propriedades desejadas para S.

No Capítulo 2, apresentaremos alguns esquemas de *hashing* perfeitos e mínimos que são algoritmos aleatorizados. Dizemos que um esquema de *hashing* é *aleatorizado* quando algumas de suas instruções são sorteios. Numa definição alternativa, mas equivalente, um esquema de *hashing* é aleatorizado quando é alimentado por *bits* aleatórios.

Quando executamos um algoritmo aleatorizado para uma mesma entrada várias vezes, é possível que obtenhamos uma saída diferente para cada execução. Em contrapartida, um algoritmo é dito *determinístico* quando, se alimentado por uma mesma entrada, sempre computa a mesma saída. Chamamos de *desaleatorização* o processo de remover os sorteios de um algoritmo aleatorizado para, a partir dele,

obter um algoritmo determinístico.

Alguns algoritmos aleatorizados podem não apenas dar uma resposta diferente para cada execução como podem também dar uma resposta errada com uma certa probabilidade limitada. No entanto, esse não é o caso de nenhum dos algoritmos de *hashing* que apresentamos neste trabalho. Frequentemente, os termos *algoritmo aleatorizado* e *algoritmo probabilístico* são apresentados na literatura como sinônimos. No entanto, para que não se confundam nossos algoritmos com aqueles que podem dar uma resposta errada, conhecidos como *algoritmos de Monte Carlo*, preferimos lhes atribuir o título apenas de aleatorizados. A saber, algoritmos aleatorizados que nunca dão resposta errada são conhecidos como *algoritmos de Las Vegas*.

1.2.1 Cotas inferiores de tempo para esquemas perfeitos e mínimos

Se o desejável é que as *hashes* sejam funções computáveis em tempo $O(1)$, o desejável para os esquemas de *hashing* perfeitos é que sejam computáveis em tempo $O(n)$. É intuitivo que não poderíamos construir um esquema executável em tempo $o(n)$, uma vez que ele precisaria, no mínimo, ler todas as chaves para gerar a *hash* perfeita. Isso significa que $\Omega(n)$ é, como mostramos a seguir, uma cota inferior de tempo para os esquemas de *hashing* perfeitos — e, consequentemente, para os esquemas de *hashing* perfeitos e mínimos.

TEOREMA 1.16: *Todo esquema de hashing perfeito é computável em tempo assintoticamente não menos que linear no número de chaves, assumindo que o universo é suficientemente maior que o conjunto de chaves.*

Demonstração: Seja n o número de chaves, suficientemente menor que o número de elementos no universo, e suponhamos que exista um esquema de *hashing* E cuja complexidade de tempo seja $O(n)$ mas não $\Omega(n)$. Seja S_1 um conjunto particular de chaves tal que $|S_1| = n$, e consideremos uma execução particular de E para a entrada S_1, a qual constrói a *hash* h_1, perfeita para S_1. Caso o esquema tenha feito sorteios ao longo dessa execução, sejam $b_1, \ldots, b_t$ os *bits* sorteados, para algum $t > 0$.

Na execução em questão, como a complexidade de tempo de E não é $\Omega(n)$, para todo n suficientemente grande há pelo menos uma chave $x_1 \in S_1$ que não foi lida e, portanto, não consultada para que a *hash* h_1 fosse estabelecida, assim como há pelo menos um $x_2 \notin S_1$ tal que $h_1(x_2) \neq h_1(x_1)$ mas $h_1(x_2) = h_1(x_3)$ para algum $x_3 \in S_1$. Tomemos $S_2 = (S_1 \setminus \{x_1\}) \cup \{x_2\}$ e a execução de E para S_2 que, caso sorteie *bits*, sorteie os mesmos $b_1, \ldots, b_t$. Evidentemente, x_2 não é lida nesta execução, e a *hash* perfeita que o esquema constrói para S_2 ainda é h_1. Ora, $h_1(x_2) = h_1(x_3)$, e ambos $x_2, x_3 \in S_2$. Logo, h_2 não é perfeita para S_2, o que é um absurdo. □

No Capítulo 2, apresentaremos alguns esquemas de *hashing* computáveis em tempo $O(n)$, mostrando que a cota proposta pelo Teorema 1.16 é justa.

1.2.2 Cotas inferiores de espaço para esquemas perfeitos e mínimos

Como uma *hash* h é uma função de U em M, uma maneira de representar h seria listar explicitamente as imagens de cada $a \in U$ por h. Não obstante, representar h desse modo custaria $\Omega(u)$ *bits*, o que seria muito indesejável. Evidentemente, há maneiras muito mais eficientes de se representar uma *hash*, e elas variam de esquema para esquema. Por exemplo, sendo uma constante $a \in U$, a *hash* $h\colon x \to ax \bmod m$,

lembrando que $M = \{0, \ldots, m-1\}$, precisa apenas que armazenemos a constante a, o que nos custaria $\lfloor \log u \rfloor + 1$ *bits*. Note-se, contudo, que, diferentemente da abordagem clássica de Análise de Espaço de Algoritmos, quando estudamos o espaço de um esquema de *hashing*, não tratamos do espaço utilizado durante a execução do esquema, outrossim abordamos apenas o espaço necessário para se representar a *hash* construída, após encerrada a execução do esquema.

Dado um universo U, a complexidade de cada $S \subseteq U$ é o tamanho da representação da melhor *hash* construída por algum esquema perfeito e mínimo que recebe S como entrada. Por *melhor*, entenda-se uma *hash* cuja representação de tamanho t é tal que nenhum outro esquema perfeito e mínimo fornece para S uma *hash* com tamanho de representação menor que t. Note-se que, neste contexto, a melhor *hash* não é necessariamente única. Em contrapartida, a complexidade de um esquema E perfeito e mínimo para um determinado n é o tamanho da representação da pior *hash* que E constrói para algum S com n chaves. Por *pior*, entenda-se uma *hash* cuja representação de tamanho t é tal que para nenhum outro S com n chaves E constrói uma *hash* com representação maior que t. Novamente, note-se que, neste outro contexto, a pior *hash* também não é necessariamente única.

TEOREMA 1.17 (Melhorn, 1984): *Sendo U um universo qualquer de tamanho u e n um natural menor que u, existe ao menos um conjunto $S \subseteq U$ de n chaves cuja complexidade é $\Omega(n + \log\log u)$.*

Demonstração: Do Lema A.17 (p. 137), temos que existe um conjunto $S \subseteq U$ de n chaves tal que o menor tamanho de representação para uma *hash* perfeita para S é no mínimo de

$$\max\left(\frac{n(n-1)}{2m\ln 2} - \frac{n(n-1)}{2u\ln 2}\left(1 - \frac{n-1}{u}\right), \log\log u - \log\log m\right) - 1 \qquad (1.18)$$

bits, sendo m o tamanho da tabela *hash*. Ora, como o máximo de dois números reais é no mínimo a média aritmética deles, temos que esse tamanho é

$$\begin{aligned}\Omega\left(\frac{\frac{n(n-1)}{2m\ln 2} - \frac{n(n-1)}{2u\ln 2}\left(1-\frac{n-1}{u}\right) + \log\log u - \log\log m}{2} - 1\right) \\ = \Omega\Big(\frac{n(n-1)}{2m\ln 2} + \log\log u\Big),\end{aligned} \tag{1.19}$$

o que, já que $m = n$, é $\Omega(n + \log\log u)$. □

Ora, se há pelo menos um S com n chaves cuja complexidade é $\Omega(n+\log\log u)$, então, a complexidade de todo esquema perfeito e mínimo para n é $\Omega(n + \log\log u)$. Portanto, dizemos que $\Omega(n+\log\log u)$ é uma cota inferior de espaço para a saída de um esquema de *hashing* perfeito e mínimo. Ademais, trata-se de uma cota justa, já que conhecemos esquemas cujas *hashes* podem ser representadas por $O(n + \log\log u)$ *bits* para todo S de tamanho n. Neste trabalho, assumimos que $\log u = o(n)$. Assim, podemos dizer simplesmente que a cota inferior de espaço para a saída de um esquema perfeito e mínimo é simplesmente $\Omega(n)$.

Em aplicações práticas, o tamanho da representação de uma *hash* é uma questão de grande importância. Não basta apenas que uma *hash* seja representada por $O(n)$ *bits* se as constantes suprimidas pela notação assintótica forem muito grandes. O algoritmo proposto por Schmidt e Siegel (1990), por exemplo, requer no mínimo $29n$ *bits* para representar a *hash* que constrói. Isso significa que, para cada chave, no mínimo 29 *bits* serão necessários. Botelho, Pagh e Ziviani (2007) afirmam que, na prática, esse algoritmo acaba tendo um uso de espaço similar aos melhores algoritmos cujas saídas requerem representação em $O(n \log n)$ *bits*.

Fredman e Komlòs (1984) propuseram uma cota mais precisa que a apresentada no Teorema 1.17: mostraram que, se $u \geqslant n^{\alpha}$ para algum $\alpha > 2$, então, representar uma *hash* perfeita e mínima custa no mínimo $n \lg \mathrm{e} + \lg\lg u + O(\log n)$ *bits* —

usamos $\lg x$ para denotar o logaritmo de x na base 2. Em 1992, Radhakrishnan apresentou uma demonstração alternativa para um resultado equivalente. Note-se que, desprezando-se os dois últimos termos, como $\lg \mathrm{e} \cong 1{,}4427$, o resultado significa que pelo menos $1{,}44n$ *bits* são necessários para representarmos uma *hash* construída por um esquema perfeito e mínimo.

Melhorn (1984), num trabalho de grande importância teórica, mostrou um esquema que constrói *hashes* mínimas e perfeitas representáveis por no máximo $n \lg \mathrm{e} + \lg \lg u + O(\log n)$ *bits*, o que significa que a cota inferior de Fredman e Komlòs (1984) é justa. Porém, o esquema por eles proposto não é conveniente na prática, já que possui complexidade exponencial de tempo.

1.2.3 Outras definições sobre esquemas de *hashing*

Sabemos, do Teorema 1.16 (p. 31), que todo esquema de *hashing* é executável em tempo assintoticamente no mínimo linear no número de chaves n. Um esquema é dito *eficiente em tempo* quando é executável em tempo assintoticamente no máximo n, a menos de um fator multiplicativo logarítmico. Assim, consideramos eficiente em tempo mesmo um esquema executável em tempo $O(n \log n)$. Os esquemas, porém, executáveis em tempo $O(n)$, chamamos de *ótimos em tempo*.

De igual modo, um esquema é dito *eficiente em espaço* quando a *hash* que constrói é representável por um número de *bits* assintoticamente no máximo n. Consideramos eficiente em espaço mesmo um esquema que produz *hashes* representáveis por $O(n + \log n)$ *bits*, pois $\log n = O(n)$ e, por conseguinte, $O(n + \log n) = O(n)$. Os esquemas, porém, cujas *hashes* obtidas são representáveis por no máximo $n \lg e + \lg \lg u + O(\log n)$ *bits*, conforme a cota de Fredman e Komlòs (1984), chamamos

de *ótimos em espaço*.

No trabalho teórico de Hagerup e Tholey (2001), propôs-se um esquema de *hashing* perfeito e mínimo que constrói em tempo $O(n+\log\log u)$ *hashes* armazenáveis em $n \lg e + \lg\lg u + O\big(n(\log\log n)^2/\log n + \log\log\log u\big)$ *bits*, resultado muito próximo do ótimo. No entanto, na prática, além de o esquema ser de difícil implementação, o algoritmo se revela ruim para conjuntos com menos de, no mínimo, 2^{150} chaves — um número com 46 dígitos. Dizemos, portanto, que esse algoritmo não é prático. Um esquema de *hashing* é dito *prático* quando, nas aplicações reais de *Hashing*, testifica seus bons resultados teóricos.

1.3 APRESENTAÇÃO DA DISSERTAÇÃO

Um comportamento determinístico de um esquema de *hashing* é desejável por vários motivos. Esquemas aleatorizados podem fornecer uma saída diferente para uma mesma entrada a cada execução. Mas, se temos sempre a mesma saída para uma mesma entrada, estabelecemos uma função $\mathscr{H}$ cujo domínio é $\binom{U}{n}$, o conjunto de todos os conjuntos S de n chaves (Notação A.1, p. 135), e cujo contradomínio é o conjunto H das *hashes* sobre U cada uma perfeita e mínima para algum S de tamanho n. $\mathscr{H}\colon \binom{U}{n} \to H$ funcionaria ela mesma como uma *hash* sobre $\binom{U}{n}$. Desse modo, acreditamos que poderíamos construir mais facilmente melhores esquemas de *hashing* dinâmicos, inspirados nos excelentes esquemas estáticos de que dispomos. Tratamos mais desse assunto no Capítulo 4 e no Apêndice E.

Exibindo esquemas de *hashing* completamente determinísticos com resultados empíricos no mínimo equivalentes aos seus correspondentes aleatorizados, mostramos que não precisamos da aleatoridade para construirmos esquemas per-

feitos, mínimos, práticos e eficientes — ou até mesmo ótimos — em tempo e espaço. Mostramos que o comportamento indeterminado da saída para uma mesma entrada é um preço que não precisamos pagar.

Neste Capítulo, introduzimos o leitor ao assunto de *Hashing*, explicando e fundamentando a terminologia que dá título ao trabalho e mencionando algumas aplicações. No Capítulo 2, apresentamos alguns esquemas de *hashing* perfeitos e mínimos e discutimos suas outras propriedades. Tomamos os mais recentes desses esquemas e propusemos sua desaleatorização no Capítulo 3, mostrando resultados comparativos que obtivemos empiricamente e incentivando trabalhos futuros. Concluímos nosso texto no Capítulo 4, incentivando trabalhos futuros.

No Apêndice A, reunimos as demonstrações secundárias, mas necessárias, que omitimos ao longo dos capítulos. As funções *hashes* de Jenkins (1997), utilizadas por alguns esquemas abordados no Capítulo 2, apresentamos no Apêndice B. Para aqueles não familiarizados com Notação Assintótica e Análise de Algoritmos, discorremos brevemente sobre o assunto no Apêndice C. Algumas definições sobre hipergrafos, necessárias para alguns esquemas apresentados no Capítulo 2, exibimos no Apêndice D. Também esboçamos uma introdução a esquemas dinâmicos de *hashing* no Apêndice E, como motivação àqueles que desejarem estender os resultados deste trabalho.

2 RESENHA LITERÁRIA

Os esquemas que mencionamos no Capítulo 1 utilizamos apenas como exemplos para ilustrarmos os conceitos elementares de *Hashing*. Esquemas mais relevantes apresentamos neste capítulo. Começaremos apresentando o FKS, um clássico esquema de *hashing* proposto em 1984, não porque lhe aplicaremos nossa abordagem de desaleatorização, mas porque servirá como parâmetro de comparação para os outros esquemas e porque, como mencionamos no Apêndice E, já serviu de inspiração para a criação de esquemas dinâmicos de *hashing*. Os esquemas que de fato nos interessam apresentaremos nas Seções 2.2 (p. 48) e 2.4 (p. 80). São eles, respectivamente, o BMZ, de 2005, baseado numa busca gulosa em grafos, e o BDZ, de 2007, baseado numa ordenação de arestas de hipergrafos.

2.1 O ESQUEMA FKS

O esquema *FKS*, proposto por Fredman, Komlòs e Szemerédi (1984) e melhorado por Melhorn (1984), é um clássico esquema de *hashing* que constrói, em tempo $O(n^3 \log u)$, sobre um conjunto de n chaves, uma *hash* perfeita e mínima descritível por $O(n \log n)$ *bits*. Embora não seja eficiente nem em tempo nem em espaço, apresentamo-lo não apenas por inspirar modelos melhores, mas especialmente por servir de comparação para os demais algoritmos apresentados neste trabalho.

Enquanto tratarmos do FKS, assumiremos U um universo numérico

$$U = \{0, \ldots, u-1\}, \tag{2.1}$$

como acordado na Seção 1.1 (p. 25), sendo u um número primo.

2.1.1 Versão primitiva

Numa versão primitiva, o FKS, embora um esquema perfeito, não é mínimo, exigindo uma tabela *hash* de $3n$ posições. Além disso, leva tempo $O(n^2 u \log u)$ para construir uma *hash* e requer $(3n + O(1)) \lg u$ *bits* para a representar.

Fredman, Komlòs e Szemerédi (1984) mostraram que:

TEOREMA 2.2: *Dado um conjunto S com n chaves, se $m = n(n-1)+1$, então, existe um $a \in U$ não nulo para o qual a hash $h\colon U \to \{0, \ldots, m-1\}$ definida por*

$$h(x) = (ax \bmod u) \bmod m \tag{2.3}$$

é perfeita para S. Ademais, esse a pode ser encontrado em tempo $O(nu \log u)$.

Demonstração: Do Lema A.20 (p. 138), existe um $a \in U \setminus \{0\}$ tal que

$$\sum_{i=0}^{n(n-1)} \binom{|S_i|}{2} < \frac{n(n-1)}{n(n-1)+1} < 1, \tag{2.4}$$

lembrando-se que, para todo $i \in \{0, \ldots, m-1\}$, S_i é o conjunto de todas as chaves x tais que $h(x) = i$. Como

$$\sum_{i=0}^{n(n-1)} |S_i|^2 = 2 \sum_{i=0}^{n(n-1)} \binom{|S_i|}{2} + \sum_{i=0}^{n(n-1)} |S_i|, \tag{2.5}$$

e como $\sum_{i=0}^{n(n-1)} |S_i| = n$, temos que $\sum_{i=0}^{n(n-1)} |S_i|^2 < n + 2$. Se h não é perfeita, existe um S_i tal que $|S_i| \geqslant 2$, e, consequentemente, $\sum_{i=0}^{n(n-1)} |S_i|^2 \geqslant n+2$, o que é um absurdo.

Encontrar a custa, no pior caso, verificar se h é injetiva para cada $a \in U$. Como cada verificação custa tempo $O(n \log u)$, encontrar a custa $O(nu \log u)$. □

Construir um esquema de *hashing* baseado no resultado 2.2 requer que a tabela *hash* tenha tamanho quadrático no número de chaves. Contudo, podemos pensar em utilizar esse resultado não como uma *hash* perfeita para todo o conjunto S, mas como uma *hash* secundária que resolva colisões causadas por uma outra *hash* primária, cuja tabela tenha tamanho linear em n. Esta é, basicamente, a ideia do esquema que Fredman, Komlòs e Szemerédi (1984) propuseram: estabelecemos uma *hash* primária h_{-1} sobre S cujo número de colisões seja limitado superiormente e resolvemos cada colisão no endereço i com uma *hash* secundária h_i, perfeita para $S_i = \{x \in S : h_{-1}(x) = i\}$.

Limitamos o número de colisões da *hash* primária para que os tamanhos das tabelas sejam todos lineares em n. O número total de colisões de uma *hash* é dado pela soma $\sum_{i=0}^{n-1} \binom{|S_i|}{2}$, já que se trata do total de pares de chaves num mesmo *bucket* de colisão S_i para todo i. O tamanho da tabela onde serão armazenados os endereços obtidos pela composição das *hashes* primária e secundárias é, por sua vez, no máximo a soma $\sum_{i=0}^{n-1} |S_i|^2$, já que essa tabela nada mais é que a concatenação das tabelas das *hashes* secundárias, cada uma possuindo tamanho $|S_i|(|S_i| - 1) + 1$. Fredman, Komlòs e Szemerédi (1984) demonstraram o seguinte resultado:

TEOREMA 2.6: *Se $m = n$, então, existe um $a \in U$ não nulo para o qual a hash $h : U \to \{0, \ldots, m-1\}$ definida pela Equação 2.3 (p. 39) é tal que $\sum_{i=0}^{n} |S_i|^2 < 3n$.*

Demonstração: Do Lema A.20 (p. 138), existe um $a \in U \setminus \{0\}$ tal que

$$\sum_{i=0}^{n-1} \binom{|S_i|}{2} < n - 1. \tag{2.7}$$

Como

$$\sum_{i=0}^{n} |S_i|^2 = 2 \sum_{i=0}^{n} \binom{|S_i|}{2} + \sum_{i=0}^{n} |S_i|, \tag{2.8}$$

e como $\sum_{i=0}^{n} |S_i| = n$, temos que $\sum_{i=0}^{n} |S_i|^2 < 3n - 2$, como queríamos mostrar. □

A *hash* primária do esquema FKS é, portanto, a função

$$h_{-1} \colon U \to \{0, \ldots, n-1\} \qquad \text{definida por} \qquad h_{-1}(x) = (ax \bmod u) \bmod n, \tag{2.9}$$

sendo a como no enunciado do Teorema 2.6. Para cada $i \in \{0, \ldots, n-1\}$, a *hash* secundária perfeita para S_i, se $S_i \neq \emptyset$, é a função

$$h_i \colon U \to \{0, \ldots, c_i - 1\} \qquad \text{definida por} \qquad h_i(x) = (a_i x \bmod u) \bmod c_i, \tag{2.10}$$

sendo a_i como no enunciado do Teorema 2.2 e $c_i = |S_i|(|S_i| - 1) + 1$. A *hash* primária, portanto, fornece o índice para a tabela da *hash* secundária na qual se encontra o endereço procurado, sendo que a tabela M da *hash* composta, de tamanho suficientemente $3n$ (Teorema 2.6), nada mais é que a concatenação de todas as tabelas das *hashes* secundárias. Logo, a *hash* composta é a função

$$h \colon U \to \{0, \ldots, 3n-1\} \qquad \text{definida por} \qquad h(x) = C_{h_{-1}(x)} + h_{h_{-1}(x)}(x), \tag{2.11}$$

sendo $C_i = \sum_{j=0}^{i-1} c_j$ o deslocamento da tabela da *hash* secundária h_i na tabela M quando $S_i \neq \emptyset$. Descrevemos com mais detalhes no Algoritmo 2.12 como o FKS

primitivo funciona.

ALGORITMO 2.12 — Versão primitiva do esquema FKS.

ENTRADA: Os conjuntos U, M e S.

SAÍDA: A descrição de uma *hash* perfeita para S: a constante não nula $a \in U$ e os vetores $\boldsymbol{A}$, c e C, todos indexados de 0 a $n-1$.

1: De acordo com o Teorema 2.6 (p. 40), encontre a e, assim, obtenha a *hash* primária h_{-1} descrita pela Equação 2.9 (p. 41).
2: $K \leftarrow 0$.
3: PARA CADA $i \in \{0, \ldots, n-1\}$, FAÇA:
4: SE $|S_i| = 0$, ENTÃO,
5: $a_i, c_i \leftarrow 0$;
6: SENÃO,
7: $c_i \leftarrow |S_i|(|S_i|-1)+1$,
8: de acordo com o Teorema 2.2 (p. 39), encontre a_i e, assim, obtenha a *hash* secundária h_i descrita pela Equação 2.10 (p. 41),
9: FIM;
10: armazene a_i em $\boldsymbol{A}[i]$, c_i em $\boldsymbol{c}[i]$ e K em $\boldsymbol{C}[i]$;
11: $K \leftarrow K + c_i$,
12: FIM.
13: DEVOLVA a, $\boldsymbol{A}$, $\boldsymbol{c}$ e $\boldsymbol{C}$.

Como enunciado no Algoritmo 2.12, a *hash* fornecida como saída pelo esquema FKS é descrita por $(a, \boldsymbol{A}, \boldsymbol{c}, \boldsymbol{C})$, sendo:

a) a, a constante da *hash* primária (Equação 2.9);

b) $\boldsymbol{A} = (a_1, \ldots, a_n)$, sendo a_i a constante da *hash* secundária h_i (Equação 2.10) quando $S_i \neq \emptyset$, ou $a_i = 0$ quando $S_i = \emptyset$;

c) $\boldsymbol{c} = (c_1, \ldots, c_n)$, sendo $c_i = |S_i|(|S_i|-1)+1$ quando $S_i \neq \emptyset$, ou $c_i = 0$ quando $S_i = \emptyset$;

d) $\boldsymbol{C} = (C_1, \ldots, C_n)$, sendo $C_i = \sum_{j=0}^{i-1} c_j$ o deslocamento da tabela da *hash* h_i na tabela M, quando $S_i \neq \emptyset$.

Note-se que, embora cada C_i possa ser calculado a partir de $\boldsymbol{c}$, $\boldsymbol{C}$ não é dispensável. Do contrário, as buscas não poderiam ser computadas em tempo constante, pois cada busca exigiria o cálculo de um C_i. Como cada um dos vetores precisa de $3n \lg u$ *bits* para ser representado, verificamos que a *hash* precisa de $(3n + O(1)) \lg u$ *bits*.

O tempo de construção da *hash* é assintoticamente no máximo o tempo necessário para se encontrarem as $n+1$ constantes $a, a_1, \ldots, a_n$. Como cada constante pode ser encontrada em tempo $O(nu \log u)$, a versão primitiva do esquema FKS possui complexidade de tempo $O(n^2 u \log u)$.

Como exemplo, tomaremos a construção de uma *hash* perfeita para o conjunto $S = \{2, 4, 5, 15, 18, 30\} \subseteq U = \{0, \ldots, 30\}$, com $M = \{0, \ldots, 17\}$. Supomos encontrar:

a) $a = 2$, que nos fornece a *hash* primária $h_{-1} \colon U \to \{0, \ldots, 5\}$, a qual associa às chaves os endereços exibidos no Quadro 2.13;

b) $\boldsymbol{A} = (1, 0, 1, 0, 7, 3)$, que fornece as *hashes* $h_i \colon U \to \{0, \ldots, c_i - 1\}$, cada uma associando às chaves do respectivo S_i os endereços que exibimos no Quadro 2.14;

x	2	4	5	15	18	30
$h_{-1}(x) = (2x \bmod 31) \bmod 6$	4	2	4	0	5	5

QUADRO 2.13 — Uma *hash* primária do FKS primitivo.

i	S_i	c_i	C_i	a_i	$h_i(x)$	$h_i(S_i)$
0	$\{15\}$	1	0	1	$(1x \bmod 31) \bmod 1$	$\{0\}$
1	$\emptyset$	0	1	0	—	—
2	$\{4\}$	1	1	1	$(1x \bmod 31) \bmod 1$	$\{0\}$
3	$\emptyset$	0	2	0	—	—
4	$\{2, 5\}$	3	2	7	$(7x \bmod 31) \bmod 3$	$\{h_4(2) = 2, h_4(5) = 1\}$
5	$\{18, 30\}$	3	5	3	$(3x \bmod 31) \bmod 3$	$\{h_5(18) = 2, h_5(30) = 1\}$

QUADRO 2.14 — *Hashes* secundárias do FKS primitivo.

Assim, obtemos a *hash* composta $h\colon U \to \{0, \ldots, 17\}$ mostrada no Quadro 2.15.

x	2	4	5	15	18	30
$i = h_{-1}(x)$	4	2	4	0	5	5
$h_i(x)$	2	0	1	0	2	1
C_i	2	1	2	0	5	5
$h(x)$	4	1	3	0	7	6

QUADRO 2.15 — Uma *hash* construída pelo FKS primitivo.

2.1.2 Versão clássica

A versão primitiva do FKS não é mínima, já que exige $3n$ posições na tabela *hash* M para um conjunto com n chaves. Veremos agora como reduzir o tamanho de M de $3n$ para n, mantendo a *hash* perfeita e tornando-a também mínima. A complexidade de tempo do esquema se reduz de $O(n^2 u \log u)$ para $O(n^3 \log u)$. O espaço, em *bits*, de representação da *hash* construída, antes $O(n \log u)$, agora se torna $(12n + 8)n \lg n + \lg \lg u + O(1) = O(n \log n + \log \log u)$, que pode ser considerado $O(n \log n)$ se assumirmos $\log u = o(n)$.

Fredman, Komlòs e Szemerédi (1984) conseguiram substituir os indesejáveis fatores u nas complexidades graças a uma *hash* ζ perfeita para um conjunto de chaves S, mediante a qual se *troca* o universo $\{0, \ldots, u-1\}$ por um outro mais conveniente, com perto de n^2 elementos. O resultado do qual os autores extraíram a *hash* ζ transcrevemos no Teorema 2.16.

TEOREMA 2.16: *Dado um $S \subseteq U$, $|S| = n$, existe um primo $q < n^2 \ln u$ tal que a hash $\zeta\colon U \to \{0, \ldots, q-1\}$ definida por $\zeta(x) = x \bmod q$ é perfeita para S.*

Demonstração: Seja $S = \{x_1, \ldots, x_n\}$ e seja

$$t = \left(\prod_{\substack{i,j \in \{1,\ldots,n\} \\ i<j}} (x_i - x_j) \right) \left(\prod_{i \in \{1,\ldots,n\}} x_i \right). \tag{2.17}$$

Notemos que

$$|t| \leqslant \left(\prod_{\substack{i,j \in \{1,\ldots,n\} \\ i<j}} u \right) \left(\prod_{i \in \{1,\ldots,n\}} u \right) = u^{\binom{n}{2}} \cdot u^n = u^{\binom{n+1}{2}} \tag{2.18}$$

e que, consequentemente, $\ln |t| \leqslant \binom{n+1}{2} \ln u$. Do Teorema dos Números Primos, exposto, por exemplo, em Graham, Knuth e Patashnik (1994, cap. 4),

$$\ln \left(\prod_{\substack{q<x \\ q \text{ primo}}} q \right) = x + o(x). \tag{2.19}$$

Assim, se todo primo q menor que $n^2 \ln u$ divide t, então,

$$\prod_{\substack{q<n^2 \ln u \\ q \text{ primo}}} q < t, \tag{2.20}$$

e, dessarte, $n^2 \ln u + o(n^2 \ln u) < \binom{n+1}{2} \ln u$, o que é impossível. Portanto, existe algum q menor que $n^2 \ln u$ que, por não dividir t, não divide nenhum x_i e nenhum $x_i - x_j$, para quaisquer distintos $x_i, x_j \in S$. Para esse q, a *hash* $\zeta \colon U \to \{0, \ldots, q-1\}$ definida por $\zeta(x) = x \bmod q$ é perfeita para S. □

Combinando o resultado do Teorema 2.16 com o do Teorema 2.2 (p. 39), temos que é possível encontrar um primo $q < n^2 \ln u$ e um $a_\rho < q$ tais que a *hash* $\rho \colon U \to \{0, \ldots, n^2 - 1\}$ definida por $\rho(x) = (a_\rho x \bmod q) \bmod n^2$ é perfeita para S. Por

isso, antes de mais nada, trocaremos o universo U por $U' = \{0, \ldots, p-1\}$, sendo p o primeiro primo maior que n^2, e trocaremos o conjunto de chaves S por $S' = \rho(S)$. Assim, poderemos substituir, nas análises de complexidade, os termos em $O(\log u)$ por $O(\log p) = O(\log n)$.

Construir o esquema procede como na versão primitiva. Temos como *hashes* primária e secundárias, respectivamente, as funções:

$$\begin{aligned} h_{-1}\colon U' \to \{0, \ldots, n-1\} \qquad &\text{definida por} \qquad h_{-1}(x) = (ax \bmod p) \bmod n; \\ h_i\colon U' \to \{0, \ldots, c_i - 1\} \qquad &\text{definida por} \qquad h_i(x) = (a_i x \bmod p) \bmod c_i. \end{aligned} \tag{2.21}$$

Observemos que:

a) $a \in U'$ é uma constante não nula para a qual h_{-1} é tal que $\sum_{i=0}^{n} |S'_i|^2 < 3n$, sendo S'_i o *bucket* de colisão da posição i em relação a S';

b) para todo $i \in \{0, \ldots, n-1\}$, se $S'_i \neq \emptyset$, $a_i \in U'$ é uma constante não nula para a qual h_i é perfeita para S'_i, e $c_i = |S'_i|(|S'_i| - 1) + 1$;

c) para todo $i \in \{0, \ldots, n-1\}$, se $S'_i = \emptyset$, $a_i = c_i = 0$.

Evidentemente, se tomássemos a tabela *hash* M como a mera concatenação das tabelas das *hashes* h_i, precisaríamos que M tivesse $3n$ posições, e não conseguiríamos devolver uma *hash* mínima. No entanto, na concatenação das tabelas das *hashes* h_i, apenas e exatamente n posições são usadas, já que $|S'| = n$. Logo, podemos tomar M com n posições se guardarmos num vetor $\boldsymbol{P}$, com $3n$ posições, indexadas de 0 a $3n-1$, os índices de M. Assim, se a uma chave x seria associado o endereço i pela *hash* composta, agora, será associado o endereço $\boldsymbol{P}[i]$. Por fim, a *hash* perfeita e mínima que o esquema FKS constrói para um conjunto S com n chaves de um universo U é a função

$$h\colon U \to \{0, \ldots, n-1\} \qquad \text{definida por} \qquad h(x) = \boldsymbol{P}[C_{h_{-1}(\rho(x))} + h_{h_{-1}(\rho(x))}(\rho(x))], \tag{2.22}$$

sendo $C_i = \sum_{j=0}^{i-1} c_j$.

Uma *hash* h construída pelo esquema FKS (Algoritmo 2.23) é descrita por $(q, a_\rho, p, a, \boldsymbol{A}, \boldsymbol{c}, \boldsymbol{C}, \boldsymbol{P})$. q e a_ρ precisam, cada um, de $2\lg n + \lg\lg u + \lg\lg \mathrm{e}$ *bits*; p e a, apenas de $2\lg n$ cada. Cada um dos vetores precisa de $2n\lg n$ *bits*, à exceção de $\boldsymbol{P}$, que precisa de $6n\lg n$ *bits*. No total, temos, como prometido, que h precisa de $(12n+8)n\lg n + \lg\lg u + O(1)$ *bits*, o que pode ser considerado $O(n\log n)$ *bits* se assumirmos $\log u = o(n)$. As constantes q e a_ρ podem ser encontrados em tempo $O(n^3 \log u)$, e cada constante $a, a_1, \ldots, a_n$ pode ser encontrada em tempo $O(n^2 \log u)$. Portanto, temos que $O(n^3 \log u)$ é de fato a complexidade do tempo de construção de esquema.

ALGORITMO 2.23 — O esquema FKS.

ENTRADA: Os conjuntos U e S.
SAÍDA: A descrição de uma *hash* perfeita e mínima para S: $(q, a_\rho, p, a, \boldsymbol{A}, \boldsymbol{c}, \boldsymbol{C}, \boldsymbol{P})$.

Encontre q, a_ρ e, assim, obtenha ρ.
Encontre p.
Encontre a e, assim, obtenha h_{-1}.
$K \leftarrow 0$.
PARA CADA $i \in \{0, \ldots, n-1\}$, FAÇA:
  SE $|S'_i| = 0$, ENTÃO,
    $a_i, c_i \leftarrow 0$;
  SENÃO,
    $c_i \leftarrow |S'_i|(|S'_i| - 1) + 1$;
    encontre a_i e, assim, obtenha h_i,
  FIM;
  armazene a_i em $\boldsymbol{A}[i]$, c_i em $\boldsymbol{c}[i]$ e K em $\boldsymbol{C}[i]$;
  $K \leftarrow K + c_i$,
FIM.
$\ell \leftarrow 0$.
PARA CADA $x \in S$, FAÇA:
  $P[C_{h_{-1}(\rho(x))} + h_{h_{-1}(\rho(x))}(\rho(x))] \leftarrow \ell$;
  $\ell \leftarrow \ell + 1$,
FIM.
DEVOLVA q, a_ρ, p, a, $\boldsymbol{A}$, $\boldsymbol{c}$, $\boldsymbol{C}$ e $\boldsymbol{P}$.

2.2 O ESQUEMA BMZ

Czech, Havasb e Majewski (1997) propuseram um esquema de *hashing* perfeito, mínimo, prático e eficiente em tempo, mas não em espaço, que utiliza grafos aleatórios acíclicos com cn vértices e n arestas, sendo c uma constante no mínimo $2{,}09$. Para cada chave $x \in S$, dois vértices distintos no grafo G são computados: $h_1(x)$ e $h_2(x)$. Assim, as chaves são associadas às arestas do grafo, antes de ser construída a *hash* h.

$$E(G) = \{\{h_1(x), h_2(x)\} : x \in S\}. \tag{2.24}$$

Para os leitores não familiarizados com Teoria dos Grafos, recomendamos o livro de Diestel (2000).

A ideia básica desse esquema é sortear h_1 e h_2 até que o grafo gerado seja acíclico, porque, se $|V| = cn$ e $c > 2$, Havas et al. (1993) mostraram que a probabilidade de G ser acíclico é

$$p = \mathrm{e}^{\frac{1}{c}} \sqrt{\frac{c-2}{c}}. \tag{2.25}$$

Para $c = 2{,}09$, essa probabilidade é aproximadamente $0{,}34$, e o número esperado de iterações, aproximadamente apenas $2{,}92$.

O esquema proposto por Czech, Havasb e Majewski (1997) também é muito interessante porque preserva a ordem relativa entre as chaves. Isso significa que, dadas duas chaves x e y, se $x < y$, então, $h(x) < h(y)$.

O *BMZ*, proposto por Botelho, Kohayakawa e Ziviani (2005), mas inspirado no esquema de Czech, Havasb e Majewski (1997), é um esquema de *hashing* perfeito, mínimo, prático e ótimo em tempo esperado, mas não em espaço, pois precisa de $O(n \log n)$ *bits* para a representação da *hash* construída. No BMZ, requer-se um número significativamente menor de vértices $cn = 1{,}15n$ e dispensa-se a restrição de G ser acíclico. Na verdade, as únicas restrições sobre G é que o subgrafo crítico

(Definição 2.26, p. 49) seja conexo e que o número de arestas críticas seja no máximo metade do total de arestas de G. A preservação da ordem relativa entre as chaves, no entanto, é sacrificada.

Chamamos o esquema proposto por Botelho, Kohayakawa e Ziviani (2005) de BMZ porque é com essa sigla que ele é referenciado pela biblioteca CMPH (*C Minimal Perfect Hashing Library*, mantida no `SourceForge.net` por Fabiano Botelho, Djamel Bellazzougui e Davi de C. Reis), muito embora seja chamado de BKZ por, por exemplo, Botelho, Pagh e Ziviani (2007). A sinonímia se deve ao fato de o BMZ ter sido proposto inicialmente em 2004 por Fabiano Botelho, David Menoti e Nivio Ziviani, mas publicado apenas por Botelho, Kohayakawa e Ziviani (2005).

DEFINIÇÃO 2.26 (Subgrafo crítico e subgrafo não-crítico): O subgrafo *crítico* G_{crit} dum grafo G é o subgrafo de G com grau mínimo $\delta(G_{\mathsf{crit}}) \geqslant 2$ tal que, se K é um subgrafo de G com grau mínimo $\delta(K) \geqslant 2$, K é um subgrafo de G_{crit}. O subgrafo *não-crítico* G_{ncrit} de G é o subgrafo formado pelas arestas de $E(G) \setminus E(G_{\mathsf{crit}})$ e pelos vértices que são adjacentes a essas arestas ou que são de grau 0 em G.

Estendemos a notação estabelecida para designar $E(G_{\mathsf{crit}})$ meramente por E_{crit} e $E(G_{\mathsf{ncrit}})$ meramente por E_{ncrit}, chamando também as arestas de críticas e não-críticas. Não podemos fazer o mesmo com $V(G_{\mathsf{crit}})$ e $V(G_{\mathsf{ncrit}})$ pois eles não são necessariamente disjuntos, mas usamos V_{scrit} para denotar $V(G_{\mathsf{crit}}) \cap V(G_{\mathsf{ncrit}})$ e, assim, tomamos $V(G_{\mathsf{crit}})$ por V_{crit} e $V(G_{\mathsf{ncrit}}) \setminus V_{\mathsf{scrit}}$ por V_{ncrit}, chamando seus elementos de vértices críticos e não-críticos, respectivamente.

O esquema BMZ é do tipo *MOS*. Os esquemas que utilizam a abordagem MOS, introduzida por Fox, Chen e Heath (1992), são compostos basicamente por três fases:

a) a fase de *mapeamento* (*mapping*), na qual o universo é mapeado num novo

e conveniente universo;

b) a fase de *ordenação* (*ordering*), na qual uma ordenação é estabelecida para as chaves de acordo com os endereços que lhes serão associados;

c) a fase de *busca* (*searching*), na qual ocorre a associação entre as chaves já ordenadas e os endereços.

No BMZ, a fase de mapeamento mapeia as chaves para o conjunto das arestas dum grafo. A ordenação estabelecida sobre essas arestas é a classificação delas em críticas e não-críticas (Definição 2.26). Por fim, a busca pela associação entre as chaves e os endereços é a busca por uma função de rotulação g sobre os vértices que determine a *hash*, conforme explicaremos na Equação 2.46 (p. 59). As entradas e saídas de cada uma dessas fases são descritas também no Quadro 2.27.

	Entrada	Saída
Mapeamento	S	G
Ordenação	G	G_{ncrit} e G_{crit}
Busca	G, G_{ncrit} e G_{crit}	g

QUADRO 2.27 — As fases do BMZ.

2.2.1 Fase de mapeamento

Na fase de mapeamento, o BMZ constrói um grafo G sobre um conjunto de vértices $V(G) = \{0, \ldots, t-1\}$, sendo $t = cn$ e c uma constante, e mapeia o conjunto das chaves S para o conjunto das arestas $E(G)$. Para isso, sorteia duas funções $h_1, h_2 \colon S \to V(G)$, de tal modo que

$$E(G) = \big\{\{h_1(x), h_2(x)\} \colon x \in S\big\}. \tag{2.28}$$

Vale lembrar que, até o final desta dissertação, não mais consideraremos os elementos de U como números naturais, mas como palavras de tamanho não nulo no máximo L formadas por caracteres dum alfabeto finito Σ. Assim, sendo T_1 e T_2 duas tabelas de números em $\{0, \ldots, t-1\}$, cada uma com L linhas e $|\Sigma|$ colunas, o BMZ pode obter as funções h_1 e h_2 sorteando os números em T_1 e T_2 e fazendo, para cada palavra $x = x_1 x_2 \cdots x_{|x|}$:

$$\begin{aligned} h_1(x) &= \left(\sum_{i=1}^{|x|} T_1[i, x_i] \right) \bmod t; \\ h_2'(x) &= \left(\sum_{i=1}^{|x|} T_2[i, x_i] \right) \bmod t; \\ h_2(x) &= \begin{cases} h_2'(x), & \text{se } h_2'(x) \neq h_1(x); \\ \big(2h_1(x) + 1\big) \bmod t, & \text{caso contrário.} \end{cases} \end{aligned} \tag{2.29}$$

Fazemos $h_2(x) = (2h_1(x) + 1) \bmod t$ quando $h_2'(x) = h_1(x)$ porque o grafo do esquema BMZ não pode ter laços. Também não admitimos arestas múltiplas. Assim, caso aconteça de $\{h_1(x), h_2(x)\} = \{h_1(y), h_2(y)\}$ para alguma dupla de chaves distintas x e y, simplesmente ressorteamos h_1 e h_2, iniciando-se mais uma iteração da fase de mapeamento. Como sorteamos os números das tabelas T_1 e T_2 com distribuição uniforme em $\{0, \ldots, t-1\}$, assumimos que, com a Equação 2.29, é como se estivéssemos sorteando G com distribuição suficientemente próxima da uniforme em $\mathscr{G}(\{0, \ldots, t-1\}, n)$, sendo $\mathscr{G}(V, n)$ o conjunto de todos os grafos com n arestas cujo conjunto de vértices é V.

TEOREMA 2.30: *O número esperado de iterações da fase de mapeamento do BMZ é* $\mathrm{e}^{\frac{1}{c^2}}$ *para* n *suficientemente grande.*

Demonstração: A probabilidade de $\{h_1(x), h_2(x)\} \neq \{h_1(y), h_2(y)\}$ para uma dupla

de chaves distintas x e y é a probabilidade de serem distintas n arestas sorteadas independentemente em $\binom{\{0,\ldots,t-1\}}{2}$, o conjunto de todas as $N = \binom{t}{2}$ possíveis arestas, com distribuição uniforme. Essa probabilidade é

$$p = \Big(1 - \frac{1}{N}\Big)\Big(1 - \frac{2}{N}\Big)\cdots\Big(1 - \frac{n-1}{N}\Big). \tag{2.31}$$

Assim,

$$\ln p = \ln\Bigg(\prod_{j=1}^{n-1}\Big(1 - \frac{j}{N}\Big)\Bigg) = \sum_{j=1}^{n-1}\ln\Big(1 - \frac{j}{N}\Big). \tag{2.32}$$

Ora, como $\ln(1-x) \approx -x$ para x suficientemente pequeno (Lema A.25, p. 139), temos que, para N suficientemente grande, $\ln\big(1 - \frac{j}{N}\big) \approx -\frac{j}{N}$ e, consequentemente, que

$$\ln p = \sum_{j=1}^{n-1}\Big(-\frac{j}{N}\Big) \approx -\frac{\binom{n}{2}}{N}. \tag{2.33}$$

Já que $N = \binom{t}{2} = c^2\binom{n}{2}$, podemos dizer que, para n suficientemente grande, a probabilidade de o sorteio de h_1 e h_2 servir é

$$p \approx \mathrm{e}^{-\frac{\binom{n}{2}}{N}} = \mathrm{e}^{-\frac{1}{c^2}}, \tag{2.34}$$

e, dessarte, o número esperado de iterações da fase de mapeamento até que o sorteio de h_1 e h_2 sirva é aproximadamente $\mathrm{e}^{\frac{1}{c^2}}$. □

É importante também que G_{crit} seja conexo e que $|E(G_{\mathsf{crit}})| \leqslant \frac{1}{2}|E(G)|$. Do contrário, o algoritmo não funciona. Do Teorema 2.35 (p. 53), no entanto, é certo que, se a constante c for apropriada, ambas as propriedades ocorrem com probabilidade praticamente 1, dispensando-se uma verificação. Uma vez que a constante c é propícia, o BMZ simplesmente executa o algoritmo como se fosse certeza que G_{crit} é conexo e que $|E(G_{\mathsf{crit}})| \leqslant \frac{1}{2}|E(G)|$. Caso ocorra a improbabilidade de o BMZ não conseguir

construir a *hash* na fase de busca por alguma dessas propriedades não ser satisfeita, o algoritmo é reiniciado por completo.

Dizemos que uma propriedade $\mathscr{P}$ ocorre em $\mathscr{G}(\{0,\ldots,t-1\},n)$ para quase todo grafo G se a probabilidade de $\mathscr{P}$ ocorrer para um grafo G sorteado uniformemente em $\mathscr{G}(\{0,\ldots,t-1\},n)$ tende a 1 quando $n\to\infty$. De igual modo, dizemos que uma propriedade $\mathscr{P}$ ocorre para quase nenhum $G\in\mathscr{G}(\{0,\ldots,t-1\},n)$ se a probabilidade de $\mathscr{P}$ ocorrer para um grafo G sorteado uniformemente em $\mathscr{G}(\{0,\ldots,t-1\},n)$ tende a 0 quando $n\to\infty$.

TEOREMA 2.35: *Há uma constante κ tal que, se $t<\kappa n$, as propriedades de G_{crit} ser conexo e de $|E(G_{\text{crit}})|\leqslant\frac{1}{2}|E(G)|$ ocorrem para quase nenhum $G\in\mathscr{G}(\{0,\ldots,t-1\},n)$, mas, se $t\geqslant\kappa n$, ocorrem para quase todo $G\in\mathscr{G}(\{0,\ldots,t-1\},n)$. Ademais, esse κ, sob a precisão de três casas decimais, é $1{,}152$.*

Demonstração (Esboço)*:* Segundo Pittel e Wormald (2005), se $c<2$ (o que significa que o grau médio $d(G)>1$), para quase todo $G\in\mathscr{G}(\{0,\ldots,t-1\},n)$ vale que:

$$\begin{aligned}|V(G_{\text{crit}})| &= \big(1+o(1)\big)(1-T)bt;\\ |E(G_{\text{crit}})| &= \big(1+o(1)\big)\Big((1-T)b+\frac{b(d+T-2)}{2}\Big)t.\end{aligned}\tag{2.36}$$

Na Equação 2.36, $d=d(G)$, $b=1-\frac{T}{d}$, e T é a única solução da Equação 2.37 tal que $0<T<1$:

$$T\mathrm{e}^{-T}=d\mathrm{e}^{-d}.\tag{2.37}$$

Lembrando que $t=c|E(G)|$, temos da Equação 2.36 que $\frac{|E(G_{\text{crit}})|}{|E(G)|}$ é uma função unicamente de c, já que b, d e T são todas funções de c. Logo, κ é o valor para c que torna $\frac{|E(G_{\text{crit}})|}{|E(G)|}=\frac{1}{2}$. Utilizando a Equação 2.36, Botelho, Kohayakawa e Ziviani (2005) experimentaram vários valores para c e obtiveram, com precisão de três casas

decimais, os seguintes resultados:

$$
\begin{aligned}
c = 1{,}149 &\therefore \frac{|E(G_{\mathsf{crit}})|}{|E(G)|} \cong 0{,}502;\\
c = 1{,}150 &\therefore \frac{|E(G_{\mathsf{crit}})|}{|E(G)|} \cong 0{,}501;\\
c = 1{,}151 &\therefore \frac{|E(G_{\mathsf{crit}})|}{|E(G)|} \cong 0{,}501;\\
c = 1{,}152 &\therefore \frac{|E(G_{\mathsf{crit}})|}{|E(G)|} \cong 0{,}500;\\
c = 1{,}153 &\therefore \frac{|E(G_{\mathsf{crit}})|}{|E(G)|} \cong 0{,}498.
\end{aligned}
\tag{2.38}
$$

Dessarte, concluíram que κ, sob a precisão de três casas decimais, é $1{,}152$.

Ainda da Equação 2.36, temos que para quase todo G, sob precisão de três casas, quando $c \cong \kappa$, $|E(G_{\mathsf{crit}})| \cong 0{,}500n$ e $|V(G_{\mathsf{crit}})| \cong 0{,}400n$, o que nos indica que G_{crit} é conexo também para quase todo G quando $c \cong \kappa$. □

Botelho, Kohayakawa e Ziviani (2005), para confirmar o resultado que transcrevemos no Teorema 2.35, experimentaram executar várias vezes a fase de mapeamento do BMZ para conjuntos de chaves com milhões de *URLs*, calculando $\frac{|E(G_{\mathsf{crit}})|}{|E(G)|}$ para cada grafo obtido, e, dessa forma, obter a probabilidade p aproximada de $|E(G_{\mathsf{crit}})|$ ser no máximo $\frac{1}{2}|E(G)|$. Para 4 milhões de *URLs*, eles obtiveram $p = 0$ para $c = 1{,}13; 1{,}14$ e $p = 1$ para $c = 1{,}15; 1{,}16; 1{,}17$. Por isso, no BMZ, convencionaram que c é $1{,}15$ e que, consequentemente, o grafo construído na fase de mapeamento possui $t = 1{,}15n$ vértices e n arestas.

Uma vez que, para $c = 1{,}15$, o número esperado de iterações da fase de mapeamento $\mathrm{e}^{\frac{1}{c^2}}$ é aproximadamente $2{,}13$, podemos assumir que o tempo esperado de execução da fase de mapeamento é assintoticamente no máximo o tempo de execução de cada iteração. O tempo de execução duma iteração é composto pelo tempo do preenchimento de T_1 e T_2 e pelo tempo de avaliação de h_1 e h_2 para toda chave x.

Cada sorteio em $\{0, \ldots, t-1\}$ pode ser feito em tempo $O(\log t) = O(\log n)$ Preencher todas as $2L|\Sigma|$ posições de T_1 e T_2, então, também pode ser feito em tempo $O(\log n)$, já que L e $|\Sigma|$ são constantes.

Porque os tamanhos das chaves e de T_1 e T_2 são limitados por constantes, os acessos a T_1 e T_2, assim como as operações de adição e de resto de divisão, são todos computáveis em tempo constante. Não obstante, o tempo de avaliação de h_1 e h_2 para todas as chaves é $O(n)$.

Por fim, concluímos que o tempo esperado de execução de toda a fase de mapeamento do BMZ é $O(2{,}13(n + \log n))$, que pode ser considerado $O(n)$, já que $\log n = O(n)$. Como o melhor caso da fase de mapeamento é aquele em que se sorteiam h_1 e h_2 apenas uma vez, o tempo do melhor caso também é $O(n)$. Note-se que, teoricamente, é possível que a fase de mapeamento ressorteie h_1 e h_2 indefinidamente, pois os ressorteios não são feitos necessariamente sem reposição. Assim, não há garantia de parada para o algoritmo BMZ, ainda que a probabilidade de o BMZ não parar seja exponencialmente infinitesimal, como argumentamos a seguir.

TEOREMA 2.39: *Para $c = 1{,}15$, a probabilidade de o BMZ executar k iterações da fase de mapeamento é menor que $\frac{1}{\varphi^{k-1}}$, sendo φ a razão áurea.*

Demonstração: Do Teorema 2.30 (p. 51), a probabilidade de ser necessário ressortear h_1 e h_2 é aproximadamente $1 - \mathrm{e}^{-\frac{1}{c^2}}$, o que, para $c = 1{,}15$, é aproximadamente $0{,}5305 < \frac{1}{\varphi}$. Como os ressorteios ocorrem independentemente e com reposição, a probabilidade de ocorrerem $k-1$ ressorteios, ou k iterações da fase de mapeamento, é, como queríamos mostrar, menor que $\left(\frac{1}{\varphi}\right)^{k-1}$. □

Na versão determinística do BMZ que apresentaremos no Capítulo 3, concederemos ao algoritmo garantia de parada.

2.2.2 Fase de ordenação

A fase de ordenação encontra o subgrafo crítico do grafo G obtido na fase de mapeamento. Para tanto, removem-se sucessivamente os vértices de grau 0 ou 1 até que não haja mais vértices a serem removidos. A *remoção* dum vértice v dum grafo G é o subgrafo $G - v$ obtido retirando-se de $V(G)$ o vértice v e, de $E(G)$, todas as arestas que incidem em v.

TEOREMA 2.40: *Sendo G um grafo e*

$$H(G) = \begin{cases} G, & \text{se } G \text{ é o grafo vazio ou se } \delta(G) \geqslant 2, \\ H(G - v), & \text{para algum } v \in V(G) \text{ tal que } d_G(v) \leqslant 1, \end{cases} \tag{2.41}$$

então, $H(G) = G_{\text{crit}}$.

Demonstração: É imediato que $H(G) = G_{\text{crit}} = G$ quando $\delta(G) \geqslant 2$ ou quando G é o grafo vazio. Assumamos, portanto, que $\delta(G) \leqslant 1$. Assumamos ainda, sem perda de generalidade, que não há vértices de grau 0 em G, que só há um $v \in V(G)$ de grau 1 e que ou $G - v$ é o grafo vazio ou $\delta(G - v) \geqslant 2$. Precisamos mostrar que K é subgrafo de $G - v$ para todo subgrafo K de G tal que ou K é o grafo vazio ou $\delta(K) \geqslant 2$.

Como o grafo vazio é subgrafo de qualquer grafo, assumamos que $\delta(G-v) \geqslant 2$ e tomemos um subgrafo K de G tal que $\delta(K) \geqslant 2$. Supondo, por contradição, que K não é subgrafo de $G - v$, teríamos que $v \in V(K)$, o que seria um absurdo, uma vez que $d_K(v) \leqslant d_G(v) = 1 \ngeqslant 2$. □

A fase de ordenação é descrita com mais detalhes no Algoritmo 2.42. A remoção sucessiva dos vértices de grau 1 é feita através duma pilha P. Inicialmente

(linhas 1–8), $P = \emptyset$ e $d_u = d_G(u)$ para todo $u \in V(G)$. Depois da execução das linhas 9–13, todos os vértices de grau 1 em G são empilhados em P. No laço da linha 14, desempilham-se os vértices de P até P ficar novamente vazia, e, para cada vértice v desempilhado, $d_v \leftarrow 0$ (linha 16) e d_u, para todo $u \in V(G - v)$, é atualizado para $d_{G-v}(u)$ (linhas 17–24), assim como se empilham em P aqueles vértices w que passaram a ter $d_w = 1$. Do Teorema 2.40, depois que o laço de linha 13 para, todos os vértices u que ficaram com $d_u > 0$ são os que formarão G_{crit} nas linhas 26–48.

ALGORITMO 2.42 — Fase de ordenação do BMZ.

ENTRADA: o grafo G obtido na fase de mapeamento.
SAÍDA: G_{crit} e G_{ncrit}.

```
 1: P ← ∅.
 2: PARA CADA u ∈ V(G), FAÇA:
 3:    d_u ← 0,
 4: FIM.
 5: PARA CADA {u_1, u_2} ∈ E(G), FAÇA:
 6:    d_{u_1} ← d_{u_1} + 1;
 7:    d_{u_2} ← d_{u_2} + 1,
 8: FIM.
 9: PARA CADA u ∈ V(G), FAÇA:
10:    SE d_u = 1, ENTÃO,
11:       empilhe u em P,
12:    FIM,
13: FIM.
14: ENQUANTO P ≠ ∅, FAÇA:
15:    desempilhe v de P;
16:    d_v ← 0;
17:    PARA CADA w adjacente a v, FAÇA:
18:       SE d_w > 0, ENTÃO,
19:          d_w ← d_w − 1;
20:          SE d_w = 1, ENTÃO,
21:             empilhe w em P,
22:          FIM,
23:       FIM,
24:    FIM,
25: FIM.
26: V_crit, V_ncrit, V_scrit, E_crit, E_ncrit ← ∅.
27: PARA CADA u ∈ V(G), FAÇA:
28:    SE d_u = 0, ENTÃO,
29:       V_ncrit ← V_ncrit ∪ {u};
30:    SENÃO,
31:       V_crit ← V_crit ∪ {u},
32:    FIM,
33: FIM.
34: PARA CADA v ∈ V_crit, FAÇA:
35:    SE (∃u ∈ N_G(v))(u ∈ V_ncrit), ENTÃO,
36:       V_scrit ← V_scrit ∪ {v},
37:    FIM,
38: FIM.
39: PARA CADA f = {u, v} ∈ E(G), FAÇA:
40:    SE u ∈ V_crit e v ∈ V_crit, ENTÃO,
41:       E_crit ← E_crit ∪ {f};
42:    SENÃO,
43:       E_ncrit ← E_ncrit ∪ {f},
44:    FIM,
45: FIM.
46: G_crit ← (V_crit, E_crit).
47: G_ncrit ← (V_scrit ∪ V_ncrit, E_ncrit).
48: DEVOLVA G_crit e G_ncrit.
```

A execução das linhas 1–4 e 9–12 toma tempo $O(|V(G)|)$. Computamos os graus de todos os vértices percorrendo todas as arestas (linhas 5–8), o que leva tempo $O(|E(G)|)$.

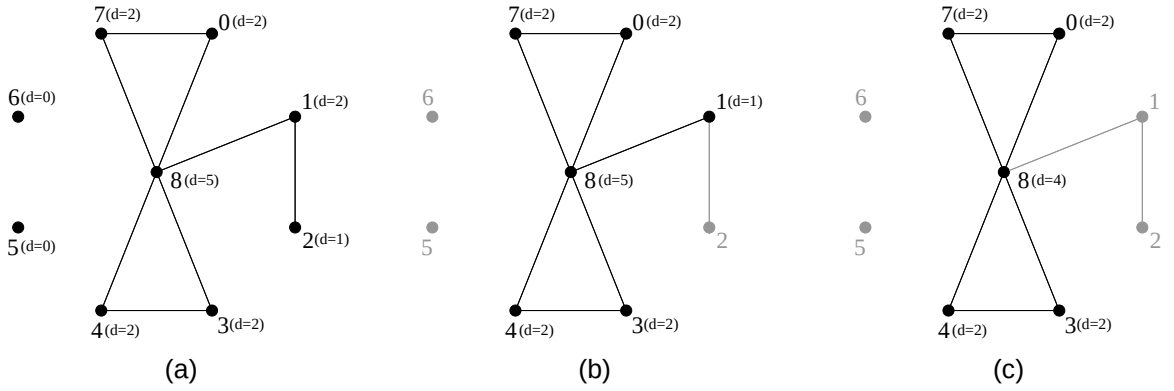
O laço da linha 14 é executado $O(|V_{\text{ncrit}}|)$ vezes, pois todo vértice que é inserido na pilha P é removido apenas uma vez e, depois disso, não torna a ser inserido; ademais, todos os vértices não-críticos, mas nenhum crítico, são inseridos na pilha.

Note-se que o tempo das linhas 14–25 é assintoticamente no máximo a soma total s do número de iterações do laço da linha 17. Não é difícil ver que $s \leqslant 2|E_{\text{ncrit}}|$, já que cada aresta incidente nalgum vértice em V_{ncrit} é verificada no mínimo 1 vez e no máximo 2 vezes e já que nenhuma aresta em E_{crit} é avaliada na linha 17.

Como as linhas 26–33 levam tempo $O(|V(G)|)$, as linhas 34–38, tempo $O(|V_{\text{crit}}|)$, e as linhas 39–45, tempo $O(|E(G)|)$, temos que o tempo total da fase de ordenação é

$$\begin{aligned} &O(|V(G)|) + O(|E(G)|) + O(|E_{\text{ncrit}}|) + O(|V(G)|) + O(|V_{\text{crit}}|) + O(|E(G)|) \\ &\quad = O(n) + O(cn) + O(n) + O(cn) + O(cn) + O(n) = O(n). \end{aligned} \tag{2.43}$$

Como exemplo, tomemos um conjunto com 8 chaves que é mapeado para o conjunto de arestas do grafo com 9 vértices representado pela Figura 2.44(a), lembrando que $8 \cdot 1{,}15 = 9{,}2 \cong 9$. Removendo-se os vértices de grau 0 ou 1, ficamos com o grafo destacado na Figura 2.44(b). Removendo-se agora o vértice 1, cujo grau se tornou 1, chegamos ao subgrafo crítico na Figura 2.44(c).



(a) (b) (c)

FIGURA 2.44 — Exemplo da fase de ordenação do BMZ.

2.2.3 Fase de busca

Como já mencionamos, o BMZ tenta na fase de busca encontrar uma rotulação dos vértices $g\colon V(G) \to \mathbb{Z}$ tal que a função $h\colon E(G) \to \mathbb{Z}$, a qual rotula as arestas, definida por

$$h(\{u_1, u_2\}) = g(u_1) + g(u_2), \tag{2.45}$$

seja uma bijeção entre $E(G)$ e $\{0, \ldots, n-1\}$.

Note-se que a função g não precisa ser necessariamente uma bijeção entre $V(G)$ e $\{0, \ldots, t-1\}$. Na realidade, ela nem o pode ser, já que $|V(G)| = t = cn > |E(G)|$ e, não obstante, as arestas e incidentes sobre vértices de rótulo maior que n teriam $h(e) > n$ também, impossibilitando h de ser bijetiva sobre $\{0, \ldots, n-1\}$. Contudo, uma vez obtida a bijeção h entre $E(G)$ e $\{0, \ldots, n-1\}$, temos nossa *hash* $\wp$ mínima e perfeita para S, dada pela Equação 2.46:

$$\wp(x) = h(\{h_1(x), h_2(x)\}), \tag{2.46}$$

sendo h_1 e h_2 as funções obtidas na fase de mapeamento (Equação 2.29, p. 51).

Não houvesse vértices críticos em G, G seria acíclico, e o procedimento para se encontrar g seria por demasiado fácil. Uma simples busca em profundidade resolveria o problema. O vértice inicial da busca em cada componente conexa receberia rótulo 0, e rotularíamos os demais vértices buscados com os valores que fossem convenientes para que as arestas buscadas recebessem, em ordem crescente, os valores de 0 até $n-1$ para h. Descrevemos com mais detalhes esse procedimento hipotético no Algoritmo 2.47 (p. 60).

ALGORITMO 2.47 — A fase de busca do BMZ caso $V(G_{\text{crit}}) = \emptyset$.

ENTRADA: Um grafo G acíclico.
SAÍDA: Uma função $g\colon V(G) \to \mathbb{Z}$ que torna a função $h\colon E(G) \to \mathbb{Z}$ definida pela Equação 2.45 (p. 59) uma bijeção entre $E(G)$ e $\{0, \ldots, n-1\}$.

1: $i \leftarrow 0$, e $P \leftarrow \emptyset$.
2: PARA CADA $u \in V(G)$, FAÇA:
3: $g(u) \leftarrow -\infty$,
4: FIM.
5: PARA CADA $u \in V(G)$ tal que $g(u) = -\infty$, FAÇA:
6: $g(u) \leftarrow 0$;
7: PARA CADA w vizinho de u tal que $g(w) = -\infty$, FAÇA:
8: empilhe $(w, g(u))$ em P,
9: FIM;
10: ENQUANTO $P \neq \emptyset$, FAÇA:
11: desempilhe (v, j) de P;
12: $g(v) \leftarrow i - j$;
13: $i \leftarrow i + 1$;
14: PARA CADA w vizinho de v tal que $g(w) = -\infty$, FAÇA:
15: empilhe $(w, g(v))$ em P,
16: FIM,
17: FIM,
18: FIM.
19: DEVOLVA g.

Por exemplo, no grafo representado pela Figura 2.48(a), começando a busca na única componente conexa pelo vértice 0, e fazendo $g(0) = 0$:

a) esperamos $h = i = 0$ para a próxima aresta buscada; portanto, ao buscarmos o próximo vértice, 1, fazemos $g(1) = 0$ e ficamos com $h(\{0, 1\}) = 0$, como queríamos (Figura 2.48(b));

b) esperamos $h = i = 1$ para a próxima aresta buscada; portanto, ao buscarmos o próximo vértice, 2, fazemos $g(2) = 1$, já que $g(1) = 0$, e ficamos com $h(\{1, 2\}) = 1$, como queríamos (Figura 2.48(c));

c) esperamos $h = i = 2$ para a próxima aresta buscada; portanto, ao buscarmos o próximo vértice, 3, fazemos $g(3) = 2$, já que $g(1) = 0$, e ficamos com $h(\{1, 3\}) = 2$, como queríamos (Figura 2.48(d));

d) esperamos $h = i = 3$ para a próxima aresta buscada; portanto, ao buscar-

mos o próximo vértice, 4, fazemos $g(4) = 1$, já que $g(3) = 2$, e ficamos com $h(\{3, 4\}) = 3$, como queríamos (Figura 2.48(e)).

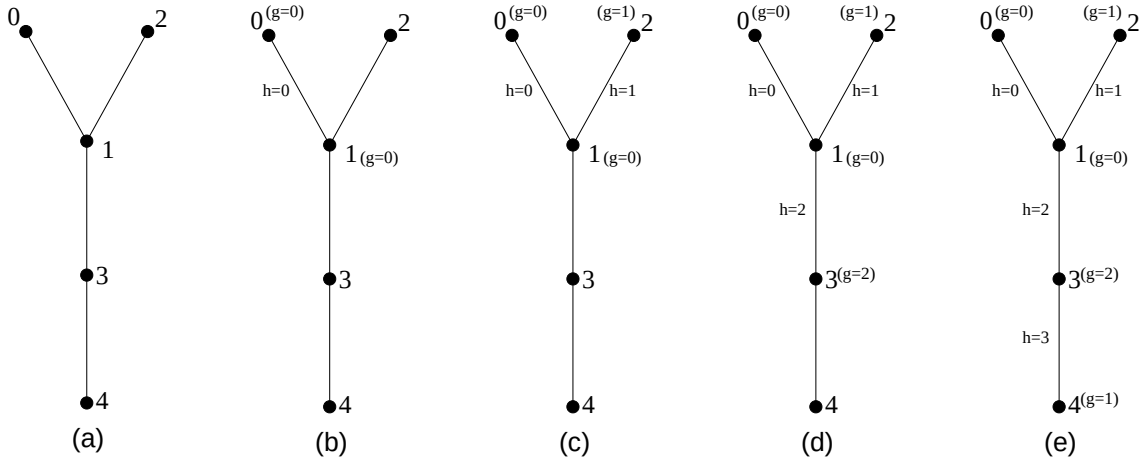


FIGURA 2.48 — Exemplo da fase de busca hipotética do BMZ.

Mostramos a seguir que esse procedimento hipotético sempre retorna uma função g que estabelece uma bijeção h entre $E(G)$ e $\{0, \ldots, n-1\}$.

TEOREMA 2.49: *É uma bijeção entre $E(G)$ e $\{0, \ldots, |E(G)|-1\}$ a função $h\colon E(G) \to \mathbb{Z}$ gerada, através da Equação 2.45 (p. 59), pela função $g\colon V(G) \to \mathbb{Z}$ obtida com a execução do Algoritmo 2.47 (p. 60) para um grafo G acíclico.*

Demonstração: Não é difícil verificar que cada aresta de G empilha um e só um vértice (linhas 7–8 e 14–15), assim como cada vértice é desempilhado só uma vez (linhas 10–11). Note-se que nem todo vértice é empilhado. Porém, ao ser desempilhado, um vértice causa um incremento de i (linha 13). O valor inicial de i é 0; assim, indutivamente, a variável i é incrementada exatas $|E(G)|$ vezes, e seu valor final é $|E(G)|$. Para todo $u \in V(G)$, $g(u) = i_u - j_u$, sendo i_u o valor de i no instante em que u é

desempilhado e

$$j_u = \begin{cases} g(v), & \text{para o vizinho } v \text{ de } u \text{ que empilhou } u \text{ (linha 8);} \\ 0, & \text{se nenhum vizinho de } u \text{ é buscado antes de } u \text{ (linha 6).} \end{cases} \tag{2.50}$$

Para toda aresta $e = \{u_1, u_2\}$ tal que u_1 é visitado antes de u_2, é u_1 quem empilha u_2 em P; pois, se u_2 é empilhado por outro vértice $u_3 \neq u_1$, isso obrigaria u_3 a ser visitado antes mesmo de u_1 e, consequentemente, geraria um caminho entre u_3 e u_1 que não passa por u_2 e, por conseguinte, um ciclo em G.

Tomemos uma aresta qualquer $e = \{u_1, u_2\}$ e assumamos, sem perda de generalidade, que u_1 é visitado antes de u_2. Uma vez que u_1 é quem empilha u_2 em P, temos que

$$h(e) = g(u_1) + g(u_2) = g(u_1) + i_{u_2} - g(u_1) = i_{u_2}. \tag{2.51}$$

Como a variável i certamente será incrementada para um valor no máximo $|E(G)|$, $h(e) < |E(G)|$. Dessarte, h é sobrejetiva sobre $\{0, \ldots, |E(G)| - 1\}$.

Agora, se há duas arestas $e = \{u_1, u_2\}$ e $f = \{u_3, u_4\}$ tais que u_1 empilha u_2, u_3 empilha u_4 e $h(e) = h(f)$, então,

$$g(u_1) + g(u_2) = g(u_1) + i_{u_2} - g(u_1) = g(u_3) + g(u_4) = g(u_3) + i_{u_4} - g(u_3); \tag{2.52}$$

logo, $i_{u_2} = i_{u_4}$, e, por consequência, $u_2 = u_4$. Portanto, dado que não há ciclos em G, $u_1 = u_3$, e, enfim, $e = f$, o que nos traz que h é também injetiva. □

Uma vez que o subgrafo não-crítico de G é acíclico, vimos que tratar dos vértices não-críticos de G é fácil. Portanto, o BMZ começa a associação dos valores de g pelos vértices críticos. Depois que termina de rotular os vértices críticos, o BMZ sabe quais são os valores de h que ainda não foram utilizados e rotula os vértices

não-críticos para que as arestas não-críticas recebam esses valores, à moda do Algoritmo 2.47. O Algoritmo 2.53 apresenta a fase de busca do BMZ. Nas linhas 1–3, todo vértice é rotulado com $-\infty$. Na linha 4, a subrotina `BMZ-Busca-Parte_Crítica` (Algoritmo 2.58, p. 67) é chamada, rotulando os vértices críticos e devolvendo num vetor $\boldsymbol{A}$ os valores entre 0 e $n-1$ que não foram utilizados para rotular as arestas críticas. Por fim, na linha 5 a subrotina `BMZ-Busca-Parte_Não-Crítica` é chamada (Algoritmo 2.54, p. 64), rotulando os vértices não-críticos com os valores que sobraram em $\boldsymbol{A}$.

ALGORITMO 2.53 — Fase de busca do BMZ.

ENTRADA: G, G_{ncrit} e G_{crit}.

SAÍDA: Uma função $g\colon V(G) \to \mathbb{Z}$ que torna a função $h\colon E(G) \to \mathbb{Z}$ definida pela Equação 2.45 (p. 59) uma bijeção entre $E(G)$ e $\{0, \ldots, n-1\}$.

1: PARA CADA $u \in V(G)$, FAÇA:
2: $\quad g(u) \leftarrow -\infty$,
3: FIM.
4: $\boldsymbol{A} \leftarrow$ `BMZ-Busca-Parte_Crítica`(G, G_{crit}, g).
5: `BMZ-Busca-Parte_Não-Crítica`$(G, G_{\text{ncrit}}, g, \boldsymbol{A})$.
6: DEVOLVA g.

Deixemos a parte crítica um pouco de lado e comecemos tratando da segunda parte da fase de busca do BMZ, a parte não-crítica, a qual apresentamos no Algoritmo 2.54 (p. 64). O procedimento funciona basicamente como o procedimento hipotético que apresentamos no Algoritmo 2.47 (p. 60), em que rotulávamos um vértice não-crítico convenientemente para que a aresta que o houvesse colocado na pilha fosse associada ao valor i pela *hash* h. A diferença é que, ao invés de i iterar de 0 até $n-1$, agora i itera de 1 até $|\boldsymbol{A}|$, e usamos os valores $\boldsymbol{A}[i]$. Iniciamos a busca em profundidade pelos vértices não-críticos que são vizinhos de vértices críticos, dado que os vértices críticos já foram rotulados pela primeira parte da fase de busca. Lembremos que V_{scrit} é o conjunto dos vértices críticos que são vizinhos de vértices não-críticos.

ALGORITMO 2.54 — BMZ-Busca-Parte_Não-Crítica$(G, G_{\text{ncrit}}, g, \boldsymbol{A})$.

1: $i \leftarrow 1$, e $P \leftarrow \emptyset$.
2: PARA CADA $u \in V_{\text{scrit}}$, FAÇA:
3: PARA CADA $w \in V_{\text{ncrit}}$ vizinho de u, FAÇA:
4: empilhe $(w, g(u))$ em P,
5: FIM;
6: ENQUANTO $P \neq \emptyset$, FAÇA:
7: desempilhe (v, j) de P;
8: $g(v) \leftarrow \boldsymbol{A}[i] - j$;
9: $i \leftarrow i + 1$;
10: PARA CADA w vizinho de v, FAÇA:
11: SE $g(w) = -\infty$, empilhe $(w, g(v))$,
12: FIM,
13: FIM,
14: FIM.
15: PARA CADA u com $g(u) = -\infty$, FAÇA:
16: $g(u) \leftarrow 0$;
17: PARA CADA w vizinho de u, FAÇA:
18: empilhe $(w, g(u))$ em P,
19: FIM;
20: ENQUANTO $P \neq \emptyset$, FAÇA:
21: desempilhe (v, j) de P;
22: $g(v) \leftarrow \boldsymbol{A}[i] - j$;
23: $i \leftarrow i + 1$;
24: PARA CADA w vizinho de v, FAÇA:
25: SE $g(w) = -\infty$, empilhe $(w, g(v))$ em P,
26: FIM,
27: FIM,
28: FIM.

Mostramos a seguir que, após a execução do Algoritmo 2.54, a função g estabelece, através da Equação 2.55, uma bijeção entre $E(G_{\text{ncrit}})$ e A

$$h_{\text{ncrit}} \colon E(G_{\text{ncrit}}) \to \mathbb{Z} \qquad \text{definida por} \qquad h_{\text{ncrit}}(\{u_1, u_2\}) = g(u_1) + g(u_2), \qquad (2.55)$$

sendo A o conjunto dos valores de $\boldsymbol{A}$. Nem precisamos dizer que o número de elementos de A é o mesmo número de posições em $\boldsymbol{A}$, uma vez que assumimos que $\boldsymbol{A}$ é preenchido na parte crítica da fase de busca do BMZ com os valores *distintos* não utilizados como imagens de arestas críticas pela *hash* h procurada.

TEOREMA 2.56: *É uma bijeção entre $E(G_{\text{ncrit}})$ e A a função $h \colon E(G_{\text{ncrit}}) \to \mathbb{Z}$ gerada, através da Equação 2.55, pela função $g \colon V(G) \to \mathbb{Z}$ obtida com a execução do Algoritmo 2.54 para um grafo G.*

Demonstração: Primeiramente notemos que, após a execução do Algoritmo 2.54, g está bem definida para todo vértice de G, uma vez que assumimos que a parte crítica da busca já rotulou todos os vértices críticos e uma vez que todos os vértices não

rotulados nas linhas 2–14 são contemplados pelas linhas 15–28. Recordemos ainda que $V(G_{\text{crit}}) = V_{\text{scrit}} \cup V_{\text{ncrit}}$. Não é difícil verificar que cada aresta de G_{ncrit} empilha um e só um vértice (as arestas incidentes sobre V_{scrit} empilham vértices de V_{ncrit} nas linhas 3–4, e as arestas incidentes apenas sobre V_{ncrit} empilham vértices de V_{ncrit} nas linhas 10–11, 17–18 e 24–25), assim como cada vértice é desempilhado só uma vez — perceba-se que as linhas 4, 11, 18 e 25 só empilham vértices com $g = -\infty$ e que todo vértice desempilhado é rotulado imediatamente. Não precisamos verificar na linha 17 se w tem $g = -\infty$, pois, não fosse assim, w teria empilhado u nas linhas 2–14.

Todo vértice desempilhado causa um incremento de i (linhas 9 e 23). O valor inicial de i é 1; assim, indutivamente, a variável i é incrementada exatas $|E(G_{\text{ncrit}})|$ vezes, e seu valor final, o qual não é utilizado, é $|E(G_{\text{ncrit}})| + 1$. Assumindo que a parte crítica da fase de busca do BMZ usou exatos $|E(G_{\text{crit}})|$ valores e deixou exatos $|E(G_{\text{ncrit}})|$ valores distintos em $\boldsymbol{A}$, temos que $|A| = |E(G_{\text{ncrit}})|$.

Apenas vértices em V_{ncrit} são empilhados e, consequentemente, rotulados com valores de A pelo Algoritmo 2.54, porque os vértices de V_{scrit}, por também pertencerem a $V(G_{\text{crit}})$, já vieram do Algoritmo 2.58 (p. 67) rotulados. Porém, para todo $u \in V_{\text{ncrit}}$, $g(u) = 0$ se u foi rotulado na linha 16, ou, caso contrário, $g(u) = \boldsymbol{A}[i_u] - j_u$, sendo i_u o valor de i no instante em que u foi desempilhado e $j_u = g(v)$, para o vizinho v de u que empilhou u (linhas 4, 11, 18 e 25).

Para toda aresta $e = \{u_1, u_2\}$ tal que u_1 é visitado antes de u_2, é u_1 quem empilha u_2 em P; pois, se u_2 é empilhado por outro vértice $u_3 \neq u_1$, isso obrigaria u_3 a ser visitado antes mesmo de u_1 e, consequentemente, geraria um caminho em G_{ncrit} entre u_3 e u_1 que não passa por u_2 e um ciclo em G_{ncrit}, um absurdo, pois G_{ncrit} seguramente é acíclico.

Tomemos uma aresta qualquer $e = \{u_1, u_2\}$ e assumamos, sem perda de generalidade, que u_1 é visitado antes de u_2. Uma vez que u_1 é quem empilha u_2 em

P, temos que

$$h_{\text{ncrit}}(e) = g(u_1) + g(u_2) = g(u_1) + \boldsymbol{A}[i_{u_2}] - g(u_1) = \boldsymbol{A}[i_{u_2}]. \tag{2.57}$$

Como a variável i certamente será incrementada para um valor no máximo $|A| + 1$, $i_{u_2} \in \{1, \ldots, |A|\}$ é um índice válido para $\boldsymbol{A}$. Dessarte, h_{ncrit} é sobrejetiva sobre A.

Agora, se há duas arestas $e = \{u_1, u_2\}$ e $f = \{u_3, u_4\}$ tais que u_1 empilha u_2, u_3 empilha u_4 e $h_{\text{ncrit}}(e) = h_{\text{ncrit}}(f)$, então, $i_{u_2} = i_{u_4}$, e, por consequência, $u_2 = u_4$. Portanto, dado que não há ciclos em G_{ncrit}, $u_1 = u_3$, e, enfim, $e = f$, o que nos traz que h_{ncrit} também é injetiva sobre A. □

Uma vez rotulados os vértices críticos, vimos que é fácil rotular os vértices não-críticos: basta que lhes atribuamos os rótulos convenientes que induzam as arestas não-críticas a serem rotuladas com os valores que sobraram da rotulação das arestas críticas. Dedicar-nos-emos agora, portanto, à exibição da parte crítica da fase de busca do BMZ, conforme proposta por Botelho, Kohayakawa e Ziviani (2005).

A rotulação dos vértices críticos pode ser feita através duma estratégia gulosa sobre uma busca em largura pelos vértices críticos. Rotulamos o vértice inicial da busca com $i = 0$ e incrementamos i. A cada vértice buscado u, se não o podemos rotular com o valor corrente de i, seguimos incrementando i e *reassociando* u até obtermos um valor permitido. Chamamos de *reassociação* cada vez que incrementamos o valor de i por causa do fracasso do valor anterior em rotular u com um valor permitido.

Rotular u com algum i não é permitido quando, para algum vértice já rotulado v vizinho de u em G_{crit}, $g(v) + i$ coincide com algum valor já atribuído a alguma outra aresta qualquer de G_{crit}. Portanto, toda vez que rotulamos algum vértice u, precisamos armazenar num conjunto A_E, inicialmente vazio, todas as rotulações de arestas que

aquela rotulação de vértice causa. Fazemos isso rotulando as arestas $\{u, v\} \in E(G_{\text{crit}})$ incidentes sobre u para as quais v já esteja rotulado.

Exibimos mais detalhes desse procedimento no Algoritmo 2.58. Conforme o Teorema 2.35 (p. 53), note-se na linha 19 que, se $g(v) + i > |E(G)|$, por ocorrer a improbabilidade de $|E(G_{\text{crit}})| \not\leqslant \frac{1}{2}|E(G)|$, ou, na linha 26, por ocorrer a improbabilidade de G_{crit} não ser conexo, reiniciamos o BMZ.

ALGORITMO 2.58 — `BMZ-Busca-Parte_Crítica`(G, G_{crit}, g).

1: $i \leftarrow 0$, $F \leftarrow \emptyset$ e $A_E \leftarrow \emptyset$.
2: Tome um u_0 em $V(G_{\text{crit}})$ e enfileire u_0 em F.
3: ENQUANTO $F \neq \emptyset$, FAÇA:
4: desenfileire u de F;
5: REPITA $\{$
6: `permitido` $\leftarrow$ SIM.
7: PARA CADA $v \in N_{G_{\text{crit}}}(u)$ tal que $g(v) \neq -\infty$, sendo ainda `permitido` $=$ SIM, FAÇA:
8: SE $g(v) + i \in A_E$, ENTÃO,
9: `permitido` $\leftarrow$ NÃO;
10: $i \leftarrow i + 1$,
11: FIM,
12: FIM.
13: $\}$ ATÉ QUE `permitido` $=$ SIM;
14: $g(u) \leftarrow i$;
15: PARA CADA $v \in N_{G_{\text{crit}}}(u)$ tal que $v \notin F$, FAÇA:
16: SE $g(v) = -\infty$, ENTÃO,
17: enfileire v em F;
18: SENÃO, SE $g(v) + i > |E(G)|$, ENTÃO,
19: REINICIE o BMZ;
20: SENÃO,
21: $A_E \leftarrow A_E \cup \{g(v) + i\}$,
22: FIM,
23: FIM;
24: $i \leftarrow i + 1$;
25: FIM.
26: SE ainda há algum u em $V(G_{\text{crit}})$ tal que $g(u) = -\infty$, ENTÃO,
27: REINICIE o BMZ,
28: FIM.
29: DEVOLVA um vetor $\boldsymbol{A}$ com os valores de $\{0, \ldots, |E(G)| - 1\} \setminus A_E$.

TEOREMA 2.59: *É uma bijeção entre* $E(G_{\text{crit}})$ *e* A_E *a função*

$$h_{\text{crit}}\colon E(G_{\text{crit}}) \to \mathbb{Z} \qquad \textit{definida por} \qquad h_{\text{crit}}(\{u_1, u_2\}) = g(u_1) + g(u_2), \qquad \textit{(2.60)}$$

gerada pela função $g\colon V(G) \to \mathbb{Z}$ *obtida com a execução do Algoritmo 2.58 (p. 67) para um grafo* G*.*

Demonstração: Observemos que, após a execução do Algoritmo 2.58, g está bem definida para todo vértice em $V(G_{\text{crit}})$, pois todo vértice de $V(G_{\text{crit}})$ é enfileirado (linhas 2 e 15–17) e todo vértice desenfileirado é rotulado (linhas 4–14). Observemos ainda que todo vértice de $V(G_{\text{crit}})$ é enfileirado e desenfileirado uma única vez (linhas 2, 4 e 17), conforme o Lema A.29 (p. 140).

Se $x \in A_E$, então, x só pode ter sido adicionado a A_E numa execução da linha 21. Logo, $x = g(v) + i$ para algum $v \in V(G_{\text{crit}})$ e algum i, que, segundo a linha 14, é algum $g(u)$ para algum $u \in V(G_{\text{crit}})$ distinto de v pela linha 15. Portanto, h_{crit} é sobrejetiva.

Assumindo por contradição que h_{crit} não é injetiva, temos duas arestas distintas $\{u_1, v_1\}$ e $\{u_2, v_2\}$ tais que $g(u_1) + g(v_1) = g(u_2) + g(v_2) = x$ para algum $x \in A_E$, sendo v_2 o último vértice rotulado dentre os quatro, e tendo sido v_1 rotulado depois de u_1. Na ocasião da rotulação de v_1 na linha 14, como u_1 já estava rotulado, quando u_1 foi em seguida visitado pelo laço da linha 15, executou-se a linha 21 e x foi adicionado a A_E. Dessarte, quando da rotulação de v_2, para todo vizinho u já rotulado de v_2, incluindo u_2, verificou-se nas linhas 7–12 que $g(v_2) + g(u) \notin A_E$, um absurdo. □

Para concluirmos a corretude da fase de busca do BMZ, basta apenas que mostremos que $\max A_E \leqslant |E(G)| - 1$, como argumentamos no Teorema 2.61.

TEOREMA 2.61: *Se* $\max A_E \leqslant |E(G)| - 1$*, então, a rotulação* g *sobre* $V(G)$ *devolvida pelo Algoritmo 2.53 (p. 63) é tal que a função*

$$h\colon E(G) \to \mathbb{Z}, \qquad \textit{definida por} \qquad h(\{u_1, u_2\}) = g(u_1) + g(u_2), \qquad \textit{(2.62)}$$

é uma bijeção entre $E(G)$ *e* $\{0, \ldots, |E(G)| - 1\}$*.*

Demonstração: Como A_E e A são conjuntos disjuntos, e como $|A_E| = |E(G_{\text{crit}})|$ e $|A| = |E(G_{\text{ncrit}})|$, se $\max A_E \leqslant |E(G)| - 1$, então, $A_E \cup A = \{0, \ldots, |E(G)| - 1\}$, e, compondo h com as funções h_{crit} e h_{ncrit} definidas respectivamente pelas Equações 2.60 (p. 68) e 2.55 (p. 64), temos, da combinação dos resultados dos Teoremas 2.59 (p. 67) e 2.56 (p. 64), que h é uma bijeção entre $E(G)$ e $\{0, \ldots, |E(G)| - 1\}$. □

Botelho, Kohayakawa e Ziviani (2005) mostraram, como transcrevemos no Teorema 2.66, que $\max A_E \leqslant |E(G)| - 1$ assumindo a Conjectura 2.64, proposta por eles sem demonstração, sob a convenção da Notação 2.63.

NOTAÇÃO 2.63: Para cada $u \in V(G_{\text{crit}})$, $I(u)$ denota o número de reassociações feitas pelo Algoritmo 2.58 (p. 67) na tentativa de rotular u. Usamos N_{t} para denotar $\sum_{u \in V(G_{\text{crit}})} I(u)$. Usamos também N_{bedges} para denotar o número de arestas de retorno (Definição A.32, p. 141) em $E(G_{\text{crit}})$ segundo a busca em largura realizada pelo Algoritmo 2.58 (p. 67).

CONJECTURA 2.64 (demonstrada por nós verdadeira no Teorema 3.1 (p. 109)):

$$N_{\text{t}} \leqslant N_{\text{bedges}}. \qquad \textit{(2.65)}$$

TEOREMA 2.66: *Se vale a Conjectura 2.64, e se vale que* $|E(G_{\text{crit}})| \leqslant \frac{1}{2}|E(G)|$ *e que* G_{crit} *é conexo, vale também que* $\max A_E \leqslant |E(G)| - 1$.

Demonstração: Do Lema A.35 (p. 142),

$$\max A_E \leqslant 2|V(G_{\text{crit}})| - 3 + 2N_{\text{t}}. \tag{2.67}$$

Mas, como assumimos que $N_{\text{t}} \leqslant N_{\text{bedges}}$,

$$\max A_E \leqslant 2|V(G_{\text{crit}})| - 3 + 2N_{\text{bedges}}. \tag{2.68}$$

Porém, do Lema A.33 (p. 141), como G_{crit} é conexo,

$$N_{\text{bedges}} = |E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1. \tag{2.69}$$

Combinando as Equações 2.68 e 2.69, temos que

$$\max A_E \leqslant 2|V(G_{\text{crit}})| - 3 + 2|E(G_{\text{crit}})| - 2|V(G_{\text{crit}})| + 2 = 2|E(G_{\text{crit}})| - 1. \tag{2.70}$$

Todavia, como $|E(G_{\text{crit}})| \leqslant \frac{1}{2}|E(G)|$,

$$\max A_E \leqslant |E(G)| - 1, \tag{2.71}$$

como queríamos mostrar. □

Embora Botelho, Kohayakawa e Ziviani (2005) não tenham conseguido mostrar que $N_{\text{t}} \leqslant N_{\text{bedges}}$, verificaram empiricamente que N_{t} é sempre menor que N_{bedges} e

tende a $0{,}059n$ quando $n \geqslant 10^6$. Lembremos que, quando $c = 1{,}15$, e, por conseguinte, da Equação 2.36 (p. 53), os valores esperados para $|V(G_{\text{crit}})|$ e $|E(G_{\text{crit}})|$ são respectivamente $0{,}401n$ e $0{,}501n$, o valor esperado para $N_{\text{bedges}} = |E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1$ (Lema A.33, p. 141) é $0{,}1n + 1$, consideravelmente maior que $0{,}059n$.

Executemos a parte crítica da fase de busca do BMZ para o subgrafo crítico dum grafo G destacado da representação de G na Figura 2.72 (p. 73)(a):

a) Começando a busca pelo vértice $u_0 = 8$, tendo $i = 0$, enfileiramos e logo em seguida desenfileiramos 8. Como todos os vizinhos de 8 ainda têm $g(v) = -\infty$, 8 é rotulado com $g(8) = 0$ (Figura 2.72(b)) e todos os seus vizinhos são enfileirados, incrementando-se i: $F = [0, 3, 4, 7]$; $i = 1$; $A_E = \emptyset$.

b) 0 é o próximo vértice a ser desenfileirado. 8 é o único vizinho já rotulado de 0, mas $g(8) + i = 1$ ainda não está em $A_E = \emptyset$. Assim, 0 é rotulado com $g(0) = 1$ (Figura 2.72(c)) e, como não possui vizinhos não rotulados que já não estejam enfileirados, não enfileira vértice algum. Como 8 é o único vizinho rotulado de 0, $g(8) + i = 1$ é adicionado a A_E, incrementando-se i: $F = [3, 4, 7]$; $i = 2$; $A_E = \{1\}$.

c) 3 é o próximo vértice a ser desenfileirado. 8 é o único vizinho já rotulado de 3, mas $g(8) + i = 2$ ainda não está em $A_E = \{1\}$. Assim, 3 é rotulado com $g(3) = 2$ (Figura 2.72(d)) e, como não possui vizinhos não rotulados que já não estejam enfileirados, não enfileira vértice algum. Como 8 é o único vizinho rotulado de 3, $g(8) + i = 2$ é adicionado a A_E, incrementando-se i: $F = [4, 7]$; $i = 3$; $A_E = \{1, 2\}$.

d) 4 é o próximo vértice a ser desenfileirado. 3 e 8 são os vizinhos já rotulados de 4, mas nem $g(3) + i = 5$ nem $g(8) + i = 3$ estão em $A_E = \{1, 2\}$. Assim, 4 é rotulado com $g(4) = 3$ (Figura 2.72(e)) e, como não possui vizinhos não rotulados que já não estejam enfileirados, não enfileira vértice algum. Como 3 e 8 são os vizinhos rotulados de 4, $g(3) + i = 5$ e $g(8) + i = 3$ são

adicionados a A_E, incrementando-se i: $F = [7]$; $i = 4$; $A_E = \{1, 2, 3, 5\}$.

e) 7 é o próximo vértice a ser desenfileirado. 0 e 8 são os vizinhos já rotulados de 7, mas $g(0) + i = 5$ já está $A_E = \{1, 2, 3, 5\}$ (Figura 2.72(f)). Portanto, i é incrementado e fazemos uma reassociação. Agora, $g(0) + i = 6$ não está em $A_E = \{1, 2, 3, 5\}$, mas $g(8) + i = 5$ está (Figura 2.72(g)). Portanto, i é incrementado e fazemos uma reassociação. Agora, nem $g(0) + i = 7$ nem $g(8) + i = 6$ estão em $A_E = \{1, 2, 3, 5\}$. Assim, 7 é rotulado com $g(7) = 6$ (Figura 2.72(h)) e, como não possui vizinhos não rotulados que já não estejam enfileirados, não enfileira vértice algum. Como 0 e 8 são os vizinhos rotulados de 7, $g(0) + i = 7$ e $g(8) + i = 6$ são adicionados a A_E, incrementando-se i: $F = \emptyset$; $i = 8$; $A_E = \{1, 2, 3, 5, 6, 7\}$.

f) Como a fila está vazia, $\boldsymbol{A} = (0, 4)$ é retornado.

Agora que já rotulamos os vértices críticos, podemos rotular os não-críticos (Figura 2.73, p. 74(a)) conforme o Algoritmo 2.54 (p. 64):

a) Empilhamos todos os vizinhos não-críticos dos vértices em V_{scrit} e inicializamos i. Como $V_{\text{scrit}} = \{8\}$, $g(8) = 0$ e o único vizinho não-crítico de 8 é o 1, empilhamos $(1, 0)$: $P = [(1, 0)]$; $i = 1$; $\boldsymbol{A} = (0, 4)$.

b) Desempilhamos $(1, j = 0)$ e rotulamos 1 com $g(1) = \boldsymbol{A}[i] - j = 0$, incrementamos i e empilhamos todos os vizinhos não rotulados de 1 com o rótulo que 1 recebeu: 0 (Figura 2.73(b)). Como o único vizinho não rotulado de 1 é o 2, empilhamos $(2, 0)$: $P = [(2, 0)]$; $i = 2$; $\boldsymbol{A} = (0, 4)$.

c) Desempilhamos $(2, j = 0)$ e rotulamos 2 com $g(2) = \boldsymbol{A}[i] - j = 4$, incrementamos i e empilhamos todos os vizinhos não rotulados de 2 com o rótulo que 2 recebeu: 4 (Figura 2.73(c)). Mas 2 não têm vizinhos. Ficamos com: $P = \emptyset$; $i = 3$; $\boldsymbol{A} = (0, 4)$.

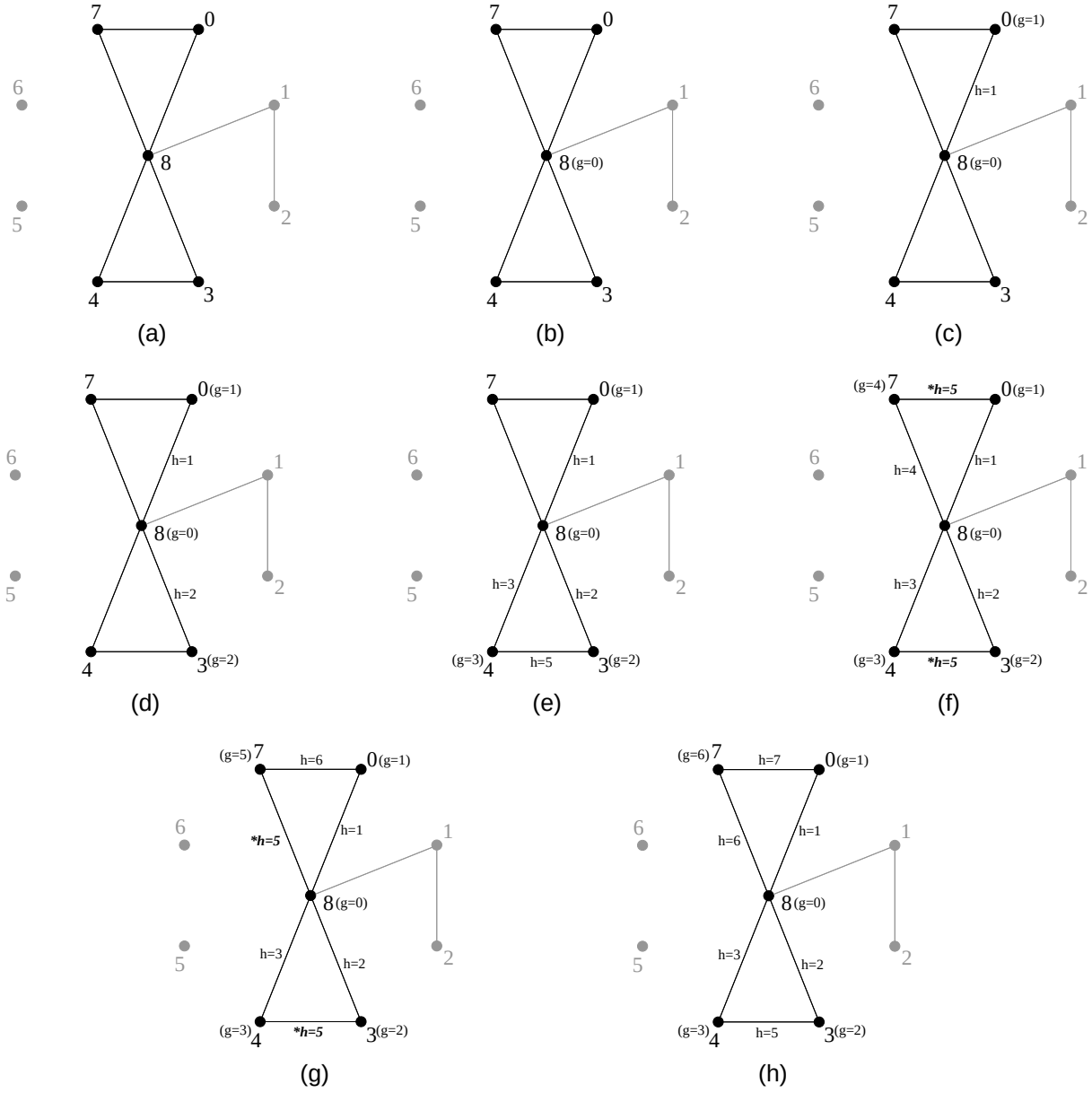


FIGURA 2.72 — Exemplo da fase de busca do BMZ — parte crítica.

d) Caímos agora no laço da linha 15, rotulando com 0 todos os vértices que não foram rotulados e empilhando os vizinhos de cada um desses vértices. Esses vértices são 5 e 6, que recebem, respectivamente, os rótulos $g(5) = 0$ e $g(6) = 0$ (Figura 2.73(d)). Mas nenhum deles possui vizinhos. Desse modo, terminamos a rotulação dos vértices de G e estabelecemos a *hash* h, como a exibimos no Quadro 2.74 (p. 75).

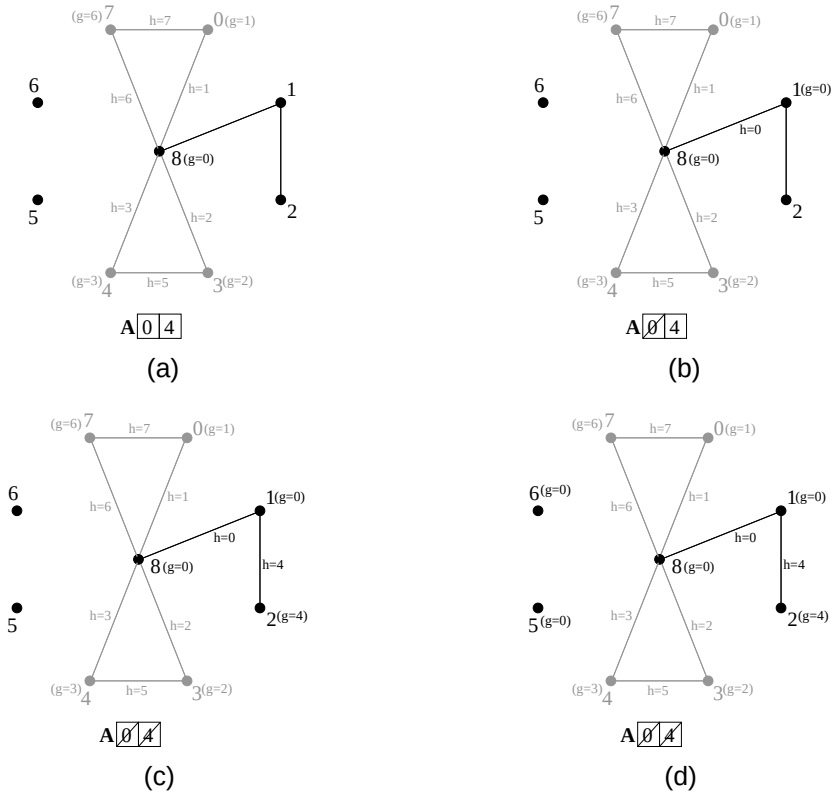


FIGURA 2.73 — Exemplo da fase de busca do BMZ — parte não-crítica.

$e \in E(G)$	$h(e)$
$\{0,7\}$	7
$\{0,8\}$	1
$\{1,2\}$	4
$\{1,8\}$	0
$\{3,4\}$	5
$\{3,8\}$	2
$\{4,8\}$	3
$\{7,8\}$	6

QUADRO 2.74 — Uma *hash* perfeita e mínima h obtida com a execução do BMZ.

2.2.4 Análise do BMZ

Como tratamos nas Seções 2.2.1 e 2.2.2, as fases de mapeamento e de ordenação do BMZ podem ser executadas juntas em tempo esperado $O(n)$. Resta-nos, todavia, discutir a complexidade de tempo da fase de busca (Algoritmo 2.53, p. 63).

Todo vértice de $V(G_{\mathsf{crit}})$ é enfileirado e desenfilerado exatamente uma vez (Lema A.29, p. 140) na parte crítica da fase de busca do BMZ (Algoritmo 2.58, p. 67). A cada vértice v desenfileirado, o laço das linhas 5–13 é iterado $I(v)+1$ vezes e, em cada iteração dessas, o laço das linhas 7–12 é iterado no máximo $d_{G_{\mathsf{crit}}}(v)$. Além disso, v ainda causa a execução de no máximo $d_{G_{\mathsf{crit}}}(v)$ iterações do laço das linhas 15–23. Portanto, a complexidade de tempo da parte crítica pode ser dada por

$$O\Bigg(\sum_{v\in V(G_{\mathsf{crit}})}\Big(\big(I(v)+1\big)d_{G_{\mathsf{crit}}}(v)+d_{G_{\mathsf{crit}}}(v)\Big)\Bigg)=O\Bigg(\sum_{v\in V(G_{\mathsf{crit}})}\big(I(v)d_{G_{\mathsf{crit}}}(v)\big)\Bigg). \tag{2.75}$$

Sendo $d(G_{\mathsf{crit}})$ o grau médio de G_{crit}, podemos assumir que $d_{G_{\mathsf{crit}}}(v)=O(d(G_{\mathsf{crit}}))$ para todo v e ficar com a seguinte complexidade de tempo para o Algoritmo 2.58:

$$O\Bigg(d(G_{\mathsf{crit}})\sum_{v\in V(G_{\mathsf{crit}})}I(v)\Bigg)=d(G_{\mathsf{crit}})N_{\mathsf{t}}. \tag{2.76}$$

Ora, uma vez que $\sum_{v \in V(G_{\mathsf{crit}})} d_{G_{\mathsf{crit}}}(v) = 2|E(G_{\mathsf{crit}})|$,

$$d(G_{\mathsf{crit}}) = \frac{2|E(G_{\mathsf{crit}})|}{|V(G_{\mathsf{crit}})|}, \tag{2.77}$$

e, da Equação 2.36 (p. 53), os valores esperados para $|V(G_{\mathsf{crit}})|$ e $|E(G_{\mathsf{crit}})|$ são respectivamente $0{,}401n$ e $0{,}501n$, o valor esperado para $d(G_{\mathsf{crit}})$ é aproximadamente $2{,}499$, uma constante. Portanto, a complexidade de tempo esperada para a execução do Algoritmo 2.58 (p. 67) é $O(N_{\mathrm{t}})$, a qual, segundo o Lema A.33 (p. 141) e a Conjectura 2.64 (p. 69) (Teorema 3.1, p. 109), é $O(|E(G_{\mathsf{crit}})|)$.

Cada vértice não-crítico de G também é empilhado e desempilhado uma só vez na parte não-crítica da fase de busca do BMZ (Algoritmo 2.54). Cada vez que um vértice v é desempilhado, quer seja na linha 7 ou na linha 21, causa a execução de no máximo $d_{G_{\mathsf{ncrit}}}(v)$ iterações do laço das linhas 10–12 ou do laço das linhas 24–26. Assim fica evidente que a complexidade de tempo da parte não-crítica da fase de busca do BMZ é

$$O\Bigg(\sum_{v \in V(G_{\mathsf{ncrit}})} d_{G_{\mathsf{ncrit}}}(v)\Bigg) = O\big(2|E(G_{\mathsf{ncrit}})|\big) = O(|E(G_{\mathsf{ncrit}})|). \tag{2.78}$$

Concluímos finalmente não apenas que a complexidade esperada da fase de busca do BMZ é de $O(|E(G_{\mathsf{crit}})| + |E(G_{\mathsf{ncrit}})|) = O(n)$, mas também concluímos que $O(n)$ é a complexidade esperada de tempo de todo o esquema BMZ para construir uma *hash* perfeita e mínima para um conjunto com n chaves.

Uma vez construída a *hash* $\wp$ dada pela Equação 2.46 (p. 59) sobre o conjunto S, computar $\wp(x)$ para uma chave $x \in S$ é algo que podemos fazer em tempo constante, pois em tempo constante conseguimos acessar as tabelas T_1 e T_2 e descobrir os vértices $h_1(x)$ e $h_2(x)$ e em tempo constante conseguimos acessar a estrutura de dados que guarda os valores $g(h_1(x))$ e $g(h_2(x))$. Fica claro que precisamos, então, para caracterizar a *hash* $\wp$, armazenar as tabelas T_1 e T_2 e todos os $1{,}15n$ valores para g, cada um requerendo $O(\log(1{,}15n))$ *bits* para ser representado. Já que as tabelas

T_1 e T_2 têm ambas tamanho constante, temos que $O(n \log n)$ é a complexidade de espaço para se representar uma *hash* construída pelo BMZ.

Mais precisamente, as tabelas T_1 e T_2 requerem, juntas, no máximo $2L|\Sigma| \lg t$ *bits*. Se as palavras são *strings* de caracteres, como *URLs*, por exemplo, $|\Sigma| \leqslant 256$, segundo a tabela ASCII. Não obstante, as tabelas T_1 e T_2 requerem, juntas, no máximo $2L(256) \lg(1{,}15n) = 103{,}237L + 512 \lg n$ *bits*.

Cada um dos $1{,}15n$ valores de g, por sua vez, precisa de no máximo $\lg(1{,}15n) \leqslant 0{,}202 + \lg n$ *bits*. Dessarte, toda a *hash* pode ser representada por no máximo

$$1{,}15n \lg n + 0{,}233n + 512 \lg n + 103{,}237L \tag{2.79}$$

bits.

2.3 O ESQUEMA J-BMZ

O BMZ, conforme apresentamos na Seção 2.2, é um esquema de *hashing* perfeito, mínimo, prático e eficiente em tempo que constrói, para n chaves, *hashes* representáveis por no máximo $1{,}15n \lg n + 0{,}233n + 512 \lg n + 103{,}237L$ *bits*. $103{,}237L + 512 \lg n$ *bits* devem-se às tabelas da Equação 2.29 (p. 51), que mapeiam as chaves para arestas dum grafo com $1{,}15n$ vértices, ao passo de que os $1{,}15n \lg n + 0{,}233n \cong (1{,}15n) \lg(1{,}15n)$ *bits* restantes devem-se aos valores dos rótulos dos vértices que determinam a *hash*, conforme as Equações 2.45 (p. 59) e 2.46 (p. 59).

Fazer o mapeamento das chaves como proposto originalmente no trabalho de Botelho, Kohayakawa e Ziviani (2005) apresenta várias desvantagens práticas:

a) as tabelas T_1 e T_2 não são pequenas e precisam ser armazenadas, adicio-

nando $103{,}237L + 512 \lg n$ *bits* à representação da *hash*;

b) preencher as tabelas custa o sorteio de $103{,}237L + 512 \lg n$ *bits*, além de que esperamos preenchê-las aproximadamente $2{,}13$ vezes (Teorema 2.30, p. 51), totalizando um número esperado de aproximadamente $219{,}895L + 1090{,}56 \lg n$ sorteios de *bits*;

c) o método de sorteio das *hashes* h_1 e h_2 pelas tabelas T_1 e T_2 não leva em conta que, na prática, frequentemente as chaves possuem muita similitude, distanciando-se muito da distribuição próxima da uniforme assumida na teoria;

d) computar cada uma das *hashes* h_1 e h_2 custa, para cada chave x de tamanho $|x|$, muitas instruções: $|x|$ somas módulo t de números com $O(\log t)$ *bits*.

As práticas *hashes* de Jenkins (1997), J_1, J_2 e J_3, apresentadas no Apêndice B, mostram-se muito convenientes para a fase de mapeamento do BMZ quando as chaves são cadeias de caracteres, como, por exemplo, *URLs*. Os três endereços de 32 *bits* cada associados a uma chave pelas três *hashes* de Jenkins são computados em paralelo por um algoritmo eficiente que faz com que cada *bit* da chave interfira em cada *bit* dos endereços. O objetivo é que, para cada *hash* de Jenkins, as chaves sejam bem distribuídas entre os 2^{32} valores na tabela *hash*, mesmo que sejam parecidas entre si ou que sigam algum padrão. Para computar os três endereços para uma chave x com $|x|$ caracteres e uma semente s, o código proposto pelo autor para implementar suas *hashes* executa apenas $6|x| + 35$ instruções.

A implementação do BMZ na biblioteca CMPH utiliza as *hashes* de Jenkins J_1 e J_2 para a fase de mapeamento, basicamente como apresentamos no Algoritmo 2.80 (p. 79). Essa versão do BMZ nós decidimos chamar de *J-BMZ*.

É importante lembrar que, quando utilizávamos as tabelas T_1 e T_2 para calcularmos h_1 e h_2, fazíamos $h_2(x) = (2h_1(x) + 1) \bmod t$, caso acontecesse de $h_2(x) =$

$h_1(x)$. Agora, Jenkins (1997) assegura que nunca ocorre de $J_1(x,s)$ ser igual a $J_2(x,s)$. Assim, $h_2(x)$ só pode coincidir com $h_1(x)$ se $J_1(x,s) \bmod t = J_2(x,s) \bmod t$.

ALGORITMO 2.80 — Fase de mapeamento do J-BMZ.

ENTRADA: um conjunto S de n chaves.
SAÍDA: um grafo G com $t = 1{,}15n$ vértices e n arestas.
1: Sorteie uma semente s de 32 *bits*.
2: PARA CADA $x \in S$, FAÇA:
3: $h_1(x) \leftarrow J_1(x,s) \bmod t$;
4: $h_2(x) \leftarrow J_2(x,s) \bmod t$;
5: SE $h_2(x) = h_1(x)$, ENTÃO,
6: $h_2(x) \leftarrow (2h_1(x) + 1) \bmod t$,
7: FIM;
8: SE $\{h_1(x), h_2(x)\} = \{h_1(y), h_2(y)\}$ para algum outro $y \in S$, ENTÃO,
9: REINICIE o algoritmo,
10: FIM,
11: FIM.
12: DEVOLVA $(\{0, \ldots, t-1\}, \{\{h_1(x), h_2(x)\} : x \in S\})$.

As fases de ordenação e busca do J-BMZ procedem exatamente como descritas nos Algoritmos 2.42 (p. 57) e 2.53 (p. 63), respectivamente. As desvantagens por conta da fase de mapeamento que havíamos discutido, no entanto, agora foram solucionadas:

a) como representar as hashes de Jenkins custa apenas 32 *bits* para a constante s que designa as *hashes*, a *hash* final do BMZ agora pode ser representada por no máximo $1{,}15n \lg n + 0{,}233n + 32$ *bits*;

b) cada iteração da fase da mapeamento custa o sorteio de apenas 32 *bits*, totalizando um número esperado de aproximadamente $2{,}13 \cdot 32 = 68{,}16$ sorteios de *bits*;

c) segundo promete o autor, as *hashes* de Jenkins distribuem bem na tabela *hash* mesmo chaves com grande similitude;

d) computar cada uma das *hashes* h_1 e h_2 agora custa, para cada chave x de tamanho $|x|$, $6|x| + 35$ instruções mais duas operações de resto de divisão

e, talvez, mais uma soma módulo t, além duma verificação simples.

2.4 O ESQUEMA BDZ

O esquema de *hashing* de Botelho, Kohayakawa e Ziviani (2005), embora seja perfeito, mínimo, prático e ótimo em tempo esperado, não é eficiente em espaço. Para um conjunto com n chaves, o esquema gera uma *hash* h que requer $O(n \log n)$ *bits* para ser representada, enquanto que, para ser considerado eficiente em espaço, um esquema precisa produzir *hashes* representáveis por $O(n)$ *bits*. O esquema que apresentamos nesta Seção, proposto por Botelho, Pagh e Ziviani (2007), não apenas é eficiente em espaço, mas também muito perto de ótimo em espaço. Chamamo-lo de *BDZ*.

O BDZ é na realidade um caso particular de uma família de esquemas de *hashing* proposta por Botelho, Pagh e Ziviani (2007). Enquanto que o BMZ mapeava as chaves para arestas de um grafo, os esquemas da família do BDZ mapeiam as chaves para arestas de um r-hipergrafo r-partido, para $r \geqslant 2$. Cada inteiro $r \geqslant 2$ determina um esquema da família, o qual chamaremos de r-BDZ. O 3-BDZ chamamos simplesmente de BDZ. Que fique claro que o r-BDZ não é uma generalização nem uma extensão do BMZ para hipergrafos. Enquanto que, no BMZ, o rótulo duma aresta era a soma dos rótulos dos vértices, no r-BDZ o rótulo duma aresta é o rótulo do vértice que designa aquela aresta. As principais definições concernentes a hipergrafos que utilizamos neste trabalho apresentamos grosseiramente no Apêndice D, embora recomendemos fortemente a leitura de Berge (1973) e de Diestel (2000, cap. 1).

Chamamos o esquema proposto por Botelho, Pagh e Ziviani (2007) de BDZ porque é com essa sigla que ele é referenciado pela biblioteca CMPH, muito embora também possa ser chamado de BPZ. A sinonímia se deve ao fato de o BDZ ter sido

concebido por Fabiano Botelho, Djamal Belazzougui, Rasmus Pagh e Nivio Ziviani, mas publicado apenas por Botelho, Pagh e Ziviani (2007).

Conforme argumentamos na Seção 1.2.2 (p. 34), a melhor cota inferior conhecida para a representação de uma *hash* construída por um esquema perfeito e mínimo é de $n \lg \mathrm{e} + \lg \lg u + O(\log n)$ *bits*. Foi proposta por Fredman e Komlòs (1984) e mostrada ser justa por Melhorn (1984). Isso significa que, para uma *hash* perfeita e mínima para n chaves, a cota inferior para o número de *bits* é de aproximadamente $1{,}44$ *bits* por chave. O BDZ é quase ótimo em espaço porque requer aproximadamente $2{,}62$ *bits* por chave. Na realidade, qualquer r-BDZ é perfeito, mínimo, ótimo em tempo e eficiente em espaço.

Nas Seções 2.4.1 (p. 82), 2.4.2 (p. 94) e 2.4.3 (p. 98), apresentaremos toda a família de esquemas do BDZ, voltando-nos aos resultados do caso particular em que $r = 3$ apenas na Seção 2.4.4.

O r-BDZ, para todo $r \geqslant 2$, também é composto por três fases. Em cada fase, estabelece-se uma função que opera injetivamente sobre a imagem da função da fase anterior. A *hash* perfeita e mínima $\wp \colon U \to \{0, \ldots, n-1\}$ (Equação 2.124, p. 98), portanto, é dada pela composição das três funções.

a) A função e (Equação 2.82, p. 83) da fase de *mapeamento* (*mapping step*) mapeia as n chaves de S para o conjunto das arestas de um r-hipergrafo r-partido G_r com $t = c(r)n$ vértices, $V(G_r) = \{0, \ldots, t-1\}$, sendo $c(r)$ a constante do r-BDZ, como definiremos na Equação 2.93 (p. 87) — cada r-BDZ possui uma constante $c(r)$ diferente. A fase de mapeamento também ordena as arestas do hipergrafo numa lista $L = [e_1, \ldots, e_n]$ de tal modo que cada aresta e_j tem ao menos um vértice que não pertence a nenhuma aresta $e_{j'}$, para todo $j' > j$.
b) A função ρ (Equação 2.113, p. 94) da fase de *designação* (*assigning step*) associa injetivamente cada aresta do hipergrafo a um de seus vértices

através duma rotulação g dos vértices.

c) A função rank (Equação 2.126, p. 98) da fase de *ranking* (*ranking step*) mapeia injetivamente para $\{0, \ldots, n-1\}$ os vértices com os quais designamos as arestas na fase anterior.

As entradas e saídas de cada uma dessas fases são descritas também no Quadro 2.81.

	Entrada	Saída
Mapeamento	S	G_r e L
Designação	G_r e L	g
Ranking	g	rank

QUADRO 2.81 — As fases do r-BDZ.

2.4.1 Fase de mapeamento

Na fase de mapeamento, a partir dum conjunto S com n chaves, construímos um r-hipergrafo r-partido G_r com $t = c(r)n$ vértices e n arestas e ainda ordenamos as arestas de G_r numa lista $L = [e_1, \ldots, e_n]$ de tal modo que toda aresta e_j tenha ao menos um vértice não pertencente às arestas que sucedem e_j em L. Se não conseguimos estabelecer essa ordenação das arestas, simplesmente ressorteamos G_r. Mostraremos no Teorema 2.92 (p. 87) que o número esperado de ressorteios é no máximo aproximadamente $2{,}76$ para $r = 2$ e de aproximadamente 0 para $r > 3$.

Construímos G_r através do sorteio de r funções, à moda da fase de mapeamento do BMZ. Cada aresta deve ter r vértices. Garantimos que G_r é r-partido tendo um vértice em $\left\{0, \ldots, \left\lfloor \frac{t}{r} \right\rfloor - 1\right\}$, um vértice em $\left\{\left\lfloor \frac{t}{r} \right\rfloor, \ldots, \left\lfloor \frac{2t}{r} \right\rfloor - 1\right\}$, um vértice em $\left\{\left\lfloor \frac{2t}{r} \right\rfloor, \ldots, \left\lfloor \frac{3t}{r} \right\rfloor - 1\right\}$, e, assim por diante, até um vértice em $\left\{\left\lfloor \frac{(r-1)t}{r} \right\rfloor, \ldots, t-1\right\}$. Logo, sorteamos r funções $h_j \colon U \to \left\{\left\lfloor \frac{jt}{r} \right\rfloor, \ldots, \left\lfloor \frac{(j+1)t}{r} \right\rfloor - 1\right\}$, sendo $0 \leqslant j < r$. Queremos que cada função h_j distribua S o máximo possível em $\left\{\left\lfloor \frac{jt}{r} \right\rfloor, \ldots, \left\lfloor \frac{(j+1)t}{r} \right\rfloor - 1\right\}$. Trataremos na Seção 2.4.4 acerca do sorteio dessas funções. Por hora, capitulamos apenas que

a função

$$e\colon S \to E(G_r) \qquad \text{definida por} \qquad e(x) = \{h_0(x), \ldots, h_{r-1}(x)\}, \tag{2.82}$$

é a função da fase de mapeamento do r-BDZ.

Note-se que, diferentemente do BMZ, para uma mesma chave x não há o perigo de $h_{j_1}(x) = h_{j_2}(x)$ se $j_1 \neq j_2$. Precisamos apenas verificar

a) se não existem chaves distintas x e y que sejam mapeadas para a mesma aresta e

b) se o hipergrafo construído G_r permite a ordenação das arestas na lista L.

Ordenamos as arestas de G_r removendo sucessivamente de G_r as arestas incidentes em vértices de grau 1 até que fiquemos com nenhuma aresta (Algoritmo 2.85, p. 84). Se, nalgum momento desse procedimento, ficamos sem vértices de grau 1 mas com arestas ainda a serem removidas, descobrimos que G_r não serve e solicitamos um novo sorteio de G_r.

Mostramos a seguir que a lista L obtida pela execução com sucesso do Algoritmo 2.85 goza da propriedade esperada.

TEOREMA 2.83: *Se o Algoritmo 2.85 (p. 84) é executado com sucesso para um hipergrafo G_r, então, na lista $L = [e_1, \ldots, e_n]$ devolvida vale que, para cada aresta e_j, existe um vértice $v_j \in e_j$ tal que $v_j \notin e_{j'}$ para todo $j' > j$.*

Demonstração: Da linha 3, toda aresta e_j é incidente num vértice v_j de grau 1 no hipergrafo $G_r^{(j)}$ definido por

$$\begin{aligned} V(G_r^{(j)}) &= V(G_r) \qquad \text{e} \\ E(G_r^{(j)}) &= E(G_r) \setminus \left(\bigcup_{i=1}^{j-1} e_i \right). \end{aligned} \tag{2.84}$$

Não obstante, e_j é a única aresta incidente sobre v_j em $G_r^{(j)}$, e, portanto, o grau de v_j em $G_r^{(j')}$, para todo $j' > j$, é 0. Assim, para todo $j' \in \{j+1, \ldots, n\}$, $v_j \notin e_{j'}$. □

ALGORITMO 2.85 — Fase de mapeamento do r-BDZ: construção de L.

ENTRADA: um hipergrafo G_r.
SAÍDA: uma ordenação $L = [e_1, \ldots, e_n]$ das arestas de G_r.
1: $j \leftarrow 1$.
2: ENQUANTO $E(G_r) \neq \emptyset$, FAÇA:
3: SE existe uma aresta $e \in E(G_r)$ incidente num vértice de grau 1, ENTÃO,
4: $e_j \leftarrow e$,
5: $E(G_r) \leftarrow E(G_r) \setminus \{e\}$, e
6: $j \leftarrow j + 1$;
7: SENÃO,
8: DEVOLVA ERRO,
9: FIM,
10: FIM.
11: DEVOLVA $L = [e_1, \ldots, e_n]$.

Como exemplo, mostramos na Figura 2.87 (p. 85) a execução do Algoritmo 2.85 para o 3-hipergrafo 3-partido definido por

$$\begin{aligned} V(G) &= \{1,2,3,4,5,6\} \qquad \text{e} \\ E(G) &= \{\{1,3,5\},\{1,3,6\},\{2,3,6\},\{2,4,5\}\}. \end{aligned} \tag{2.86}$$

Indicamos na Figura os v_j e os e_j escolhidos.

Quando um r-hipergrafo r-partido G_r é acíclico, a lista L pode ser construída com sucesso, como mostramos no Teorema 2.90. No entanto, nem todo r-hipergrafo r-partido que permite a construção de L com o Algoritmo 2.85 é acíclico. O 3-hipergrafo 3-partido G_3 representado pela Figura 2.87(a), cujo conjunto de vértices é a união disjunta $\{1,2\} \cup \{3,4\} \cup \{5,6\}$, serve para a construção de

$$L = [\{2,4,5\},\{2,3,6\},\{1,3,5\},\{1,3,6\}] \tag{2.88}$$

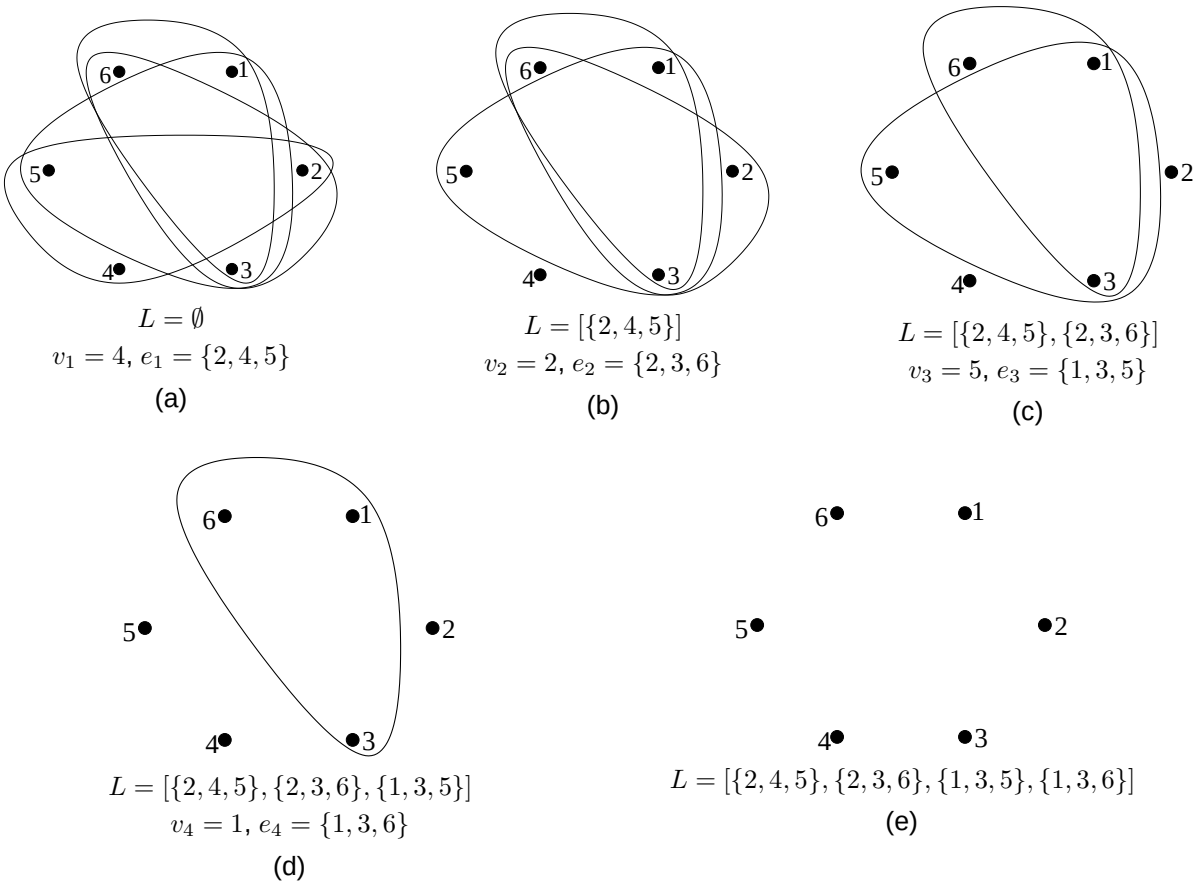


FIGURA 2.87 — Exemplo da construção de L no mapeamento do r-BDZ.

com sucesso, mas não é acíclico, pois

$$3, \{1, 3, 5\}, 5, \{2, 4, 5\}, 2, \{2, 3, 6\}, 3 \tag{2.89}$$

é um ciclo em G_r, de acordo com a Definição D.15 (p. 162).

TEOREMA 2.90: *Se um r-hipergrafo r-partido G_r é acíclico, então, a execução do Algoritmo 2.85 (p. 84) para G_r não retorna erro.*

Demonstração: Se $E(G_r) = \emptyset$, o algoritmo retorna $L = \emptyset$. Senão, do Teorema D.19 (p. 163), existe um vértice v_1 de grau 1 em G_r. Sendo e_1 a aresta incidente em v_1, como

G_r é acíclico, $G_r^{(2)}$, definido por $V(G_r^{(2)}) = V(G_r)$ e $E(G_r^{(2)}) = E(G) \setminus \{e_1\}$, também é acíclico. Indutivamente, para todo hipergrafo $G_r^{(j)}$ definido por

$$\begin{aligned} V(G_r^{(j)}) &= V(G_r) \qquad \text{e} \\ E(G_r^{(j)}) &= E(G_r) \setminus \left(\bigcup_{i=1}^{j-1} \{e_i\} \right), \end{aligned} \tag{2.91}$$

$G_r^{(j)}$ sempre possui um vértice de grau 1 e, consequentemente, a execução do Algoritmo 2.85 nunca cai na linha 8. □

Fixados t e n, temos do Teorema 2.90 que a probabilidade de um r-hipergrafo r-partido G_r com t vértices e n arestas servir para o Algoritmo 2.85 é no mínimo a probabilidade de G_r ser acíclico. Portanto, o número esperado de sorteios de um r-hipergrafo r-partido G_r com t vértices e n arestas até que G_r sirva para o Algoritmo 2.85 é no máximo o número esperado de sorteios até que G_r seja acíclico. Chamamos de iteração da fase de mapeamento cada vez que sorteamos um conjunto de *hashes* h_j para construirmos um G_r. Note-se que uma iteração pode falhar quando o sorteio das r funções h_j produz alguma aresta múltipla ou quando o hipergrafo G_r construído não serve para o Algoritmo 2.85. O Teorema 2.92 (p. 87) mostra que, fixado r, se a constante $c(r)$ que determina t for apropriada, o número esperado de iterações é uma constante: função de $c(r)$ se $r = 2$ e 1 se $r > 2$.

Botelho, Pagh e Ziviani (2007) definem *hipergrafo acíclico* como sendo um hipergrafo do qual podemos remover sucessivamente arestas incidentes em vértices de grau 1 até ficarmos com nenhuma aresta. Assim, os autores estimam o número esperado de iterações da fase de mapeamento do r-BDZ com base na probabilidade de um hipergrafo aleatório ser acíclico. Não consideramos essa definição coerente com nenhuma definição clássica de ciclo em hipergrafo. Ademais, mostramos na Figura 2.87(a) um hipergrafo cíclico do qual se podem remover sucessivamente ares-

tas incidentes em vértices de grau 1 até se ficar com nenhuma aresta. Mesmo assim, construímos a estimativa do número esperado de iterações da fase de mapeamento do r-BDZ da mesma forma que foi construída no artigo original, uma vez que, como mostramos, o número esperado de sorteios de um r-hipergrafo r-partido G_r com t vértices e n arestas até que G_r sirva para o Algoritmo 2.85 é no máximo o número esperado de sorteios até que G_r seja acíclico.

TEOREMA 2.92: *Se*

$$c(r) = \begin{cases} (2+\epsilon), \epsilon > 0, & \textit{se } r = 2, \textit{ e} \\ r\Big(\min\limits_{\delta>0}\Big\{\dfrac{\delta}{(1-e^{-\delta})^{r-1}}\Big\}\Big)^{-1}, & \textit{se } r > 2, \end{cases} \tag{2.93}$$

então, o número esperado de iterações da fase de mapeamento do r-BDZ, denotado por $\mathbb{E}I_{\textit{BDZ-map}}(r)$, é no máximo

$$\mathbb{E}I'_{\textit{BDZ-map}}(r) \approx \begin{cases} \big(\mathrm{e}^{\frac{1}{c(2)}\left(\frac{1}{c(2)}-1\right)}\big)\sqrt{\frac{2}{\epsilon}+1}, & \textit{se } r = 2, \textit{ e} \\ 1, & \textit{se } r > 2, \end{cases} \tag{2.94}$$

para n suficientemente grande.

Demonstração: Um sorteio de $h_0, \ldots, h_{r-1}$ *serve* quando o hipergrafo G_r construído não possui arestas múltiplas e não causa erro na execução do Algoritmo 2.85 (p. 84). Assim, a probabilidade p de um sorteio de $h_0, \ldots, h_{r-1}$ servir pode ser dada por

$$p(r) \geqslant p_1(r) \cdot p_2(r), \tag{2.95}$$

sendo $p_1(r)$ a probabilidade de G_r não possuir arestas múltiplas e sendo $p_2(r)$ a probabilidade de G_r ser acíclico dado que não possui arestas múltiplas.

Ora, a probabilidade $p_1(r)$ é a probabilidade de serem distintas n arestas sorteadas independentemente no conjunto de todas as N possíveis arestas, com distribuição uniforme. Essa probabilidade é

$$p_1(r) = \Big(1 - \frac{1}{N}\Big)\Big(1 - \frac{2}{N}\Big)\cdots\Big(1 - \frac{n-1}{N}\Big). \tag{2.96}$$

Não obstante,

$$\ln p_1(r) = \ln\bigg(\prod_{j=1}^{n-1}\Big(1 - \frac{j}{N}\Big)\bigg) = \sum_{j=1}^{n-1}\ln\Big(1 - \frac{j}{N}\Big). \tag{2.97}$$

Porém, como $\ln(1-x) \approx -x$ para x suficientemente pequeno (Lema A.25, p. 139), temos que, para N suficientemente grande, $\ln\big(1 - \frac{j}{N}\big) \approx -\frac{j}{N}$ e, consequentemente, que

$$\ln p_1(r) = \sum_{j=1}^{n-1}\Big(-\frac{j}{N}\Big) \approx -\frac{\binom{n}{2}}{N} \approx -\frac{n^2}{2N}. \tag{2.98}$$

Enfim, já que $N = \big(\frac{c(r)n}{r}\big)^r$,

$$p_1(r) \approx \mathrm{e}^{-\frac{r^r}{2(c(r))^r n^{r-2}}}. \tag{2.99}$$

Mas, uma vez que $\frac{1}{n^{r-2}} \to 0$ quando $n \to +\infty$ e $r > 2$, temos que, para n suficientemente grande,

$$p_1(r) \approx \begin{cases} \mathrm{e}^{-\frac{2}{(c(r))^2}}, & \text{se } r = 2, \text{ e} \\ 1, & \text{se } r > 2. \end{cases} \tag{2.100}$$

Erdös e Rényi (1960) aproximaram à distribuição de Poisson a distribuição do número de ciclos de um hipergrafo tomado aleatoriamente em $\mathscr{G}(\{0,\ldots,(2+\epsilon)n\}, n)$, quando $\epsilon > 0$, e, dessarte, mostraram que, quando $n \to +\infty$, a probabilidade de um hipergrafo com $(2+\epsilon)n$ vértices e n arestas ser acíclico tende a

$$\mathrm{e}^{\frac{1}{c}+\frac{1}{c^2}}\sqrt{\frac{c-2}{c}}, \tag{2.101}$$

sendo $c = 2 + \epsilon$. Logo,

$$p_2(2) = \mathrm{e}^{\frac{1}{c(2)} + \frac{1}{(c(2))^2}} \sqrt{\frac{\epsilon}{c(2)}}. \tag{2.102}$$

Por outro lado, Czech, Havasb e Majewski (1997, cap. 6, Teorema 6.5) mostraram que, para $r > 2$, quando $c(r)$ é como na Equação 2.93, a probabilidade de um hipergrafo com $c(r)n$ vértices e n arestas ser acíclico tende a 1 quando $t \to +\infty$. Finalmente,

$$p(r) \geqslant p_1(r) \cdot p_2(r) \approx \begin{cases} \mathrm{e}^{\frac{1}{c(2)} - \frac{1}{(c(2))^2}} \sqrt{\frac{\epsilon}{2+\epsilon}}, & \text{se } r = 2, \text{ e} \\ 1, & \text{se } r > 2. \end{cases} \tag{2.103}$$

Porque $\mathbb{E}I_{\text{BDZ-map}}(r) = \frac{1}{p(r)} \leqslant \frac{1}{p_1(r) \cdot p_2(r)} = \mathbb{E}I'_{\text{BDZ-map}}(r)$, chegamos ao resultado esperado. □

Mostramos no Quadro 2.104 alguns valores para $c(2)$ com os $\mathbb{E}I'_{\text{BDZ-map}}(2)$ obtidos conforme o resultado teórico do Teorema 2.92. Botelho, Pagh e Ziviani (2007) fizeram vários testes sobre conjuntos com 10^7 chaves e obtiveram $\mathbb{E}I_{\text{BDZ-map}}(2) \cong 3{,}401$, em média, para $c(2) = 2{,}09$ — o que implica $2{,}401$ ressorteios. Por convenção, trabalharam com $c(2) = 2{,}09$, embora nada impedisse que outros valores fossem utilizados.

$c(2)$	$\mathbb{E}I'_{\text{BDZ-map}}(2)$
2,08	3,973
2,09	3,755
2,10	3,571
2,11	3,413
2,12	3,276

QUADRO 2.104 — Alguns valores para $c(2)$ e seus consequentes $\mathbb{E}I_{\text{BDZ-map}}(2)$.

Por outro lado, para $r > 2$, temos que o real positivo δ que minizima a fração $\frac{\delta}{(1-\mathrm{e}^{-\delta})^{r-1}}$ já determina $c(r)$. Expomos no Quadro 2.105 (p. 90) com precisão de três casas decimais os valores para $c(3)$, $c(4)$ e $c(5)$.

Exibimos a fase de mapeamento do r-BDZ no Algoritmo 2.106 (p. 90). Note-se que, embora para $r > 2$ a probabilidade de G_r servir seja aproximadamente 1 para n

r	δ	$\left(\frac{\delta-0{,}001}{(1-\mathrm{e}^{-(\delta-0{,}001)})^{r-1}}; \frac{\delta}{(1-\mathrm{e}^{-\delta})^{r-1}}; \frac{\delta+0{,}001}{(1-\mathrm{e}^{-(\delta+0{,}001)})^{r-1}}\right)$	$c(r)$
3	1,256	(2,4554087; 2,4554076; 2,4554077)	1,222
4	1,904	(3,0891197; 3,0891194; 3,0891201)	1,295
5	2,337	(3,5089016; 3,5089014; 3,5089022)	1,425

QUADRO 2.105 — Alguns valores para $c(r), r > 2$, conforme o Teorema 2.92.

suficientemente grande, ainda assim verificamos essa propriedade porque é durante a execução dessa verificação que construímos a lista $L = [e_1, \ldots, e_n]$.

ALGORITMO 2.106 — Fase de mapeamento do r-BDZ.

ENTRADA: um conjunto S com n chaves.
SAÍDA: um r-hipergrafo r-partido G_r e uma ordenação $L = [e_1, \ldots, e_n]$ das arestas tal que, para todo $j \in \{1, \ldots, n\}$, existe um $v_j \in e_j$ tal que $v_j \notin e_{j'}$ para todo $j' > j$.
1: $t \leftarrow c(r)n$, sendo $c(r)$ como na Equação 2.93 (p. 87).
2: $V(G) \leftarrow \{0, \ldots, t-1\}$.
3: Sorteie r funções $h_j : U \to \left\{\left\lfloor \frac{jt}{r} \right\rfloor, \ldots, \left\lfloor \frac{(j+1)t}{r} \right\rfloor - 1\right\}, 0 \leqslant j < r$.
4: SE existem em S distintos x e y tais que $h_j(x) = h_j(y)$ para todo $j \in \{0, \ldots, r-1\}$, ENTÃO,
5: RETORNE à linha 3,
6: FIM.
7: $E(G) \leftarrow \{\{h_0(x), \ldots, h_{r-1}(x)\} : x \in S\}$.
8: $G \leftarrow (V(G), E(G))$.
9: SE a execução do Algoritmo 2.85 (p. 84) para G devolve ERRO, ENTÃO,
10: RETORNE à linha 3;
11: SENÃO, SE devolve $L = [e_1, \ldots, e_n]$, ENTÃO,
12: DEVOLVA G e L,
13: FIM.

Exemplifiquemos a execução de toda a fase de mapeamento do 3-BDZ para o conjunto de chaves $S = \{\texttt{gott}, \texttt{fried}, \texttt{wil}, \texttt{helm}, \texttt{von}, \texttt{lei}, \texttt{bniz}\}$. Como $c(3)n = 1{,}23 \cdot 7$, queremos, então, construir um 3-hipergrafo 3-partido com $t = 9$ vértices, sendo $V(G) = \{0, 1, 2\} \cup \{3, 4, 5\} \cup \{6, 7, 8\}$. Suponhamos que, na primeira iteração, sorteamos as

seguintes funções:

$$\begin{pmatrix} & h_0 \colon U \to \{0,1,2\} & h_1 \colon U \to \{3,4,5\} & h_2 \colon U \to \{6,7,8\} \\ \texttt{gott} & 0 & 3 & 6 \\ \texttt{fried} & 2 & 4 & 7 \\ \texttt{wil} & 0 & 3 & 7 \\ \texttt{helm} & 2 & 5 & 6 \\ \texttt{von} & 0 & 4 & 8 \\ \texttt{lei} & 1 & 3 & 7 \\ \texttt{bniz} & 2 & 5 & 6 \end{pmatrix} \tag{2.107}$$

Não conseguimos validar esse conjunto de funções porque $e(\texttt{helm}) = e(\texttt{bniz})$. Suponhamos, então, que na segunda iteração obtemos:

$$\begin{pmatrix} & h_0 \colon U \to \{0,1,2\} & h_1 \colon U \to \{3,4,5\} & h_2 \colon U \to \{6,7,8\} \\ \texttt{gott} & 0 & 3 & 6 \\ \texttt{fried} & 1 & 4 & 7 \\ \texttt{wil} & 1 & 5 & 8 \\ \texttt{helm} & 2 & 4 & 8 \\ \texttt{von} & 2 & 4 & 7 \\ \texttt{lei} & 1 & 4 & 8 \\ \texttt{bniz} & 0 & 4 & 7 \end{pmatrix} \tag{2.108}$$

Dessa vez, as funções sorteadas são válidas e constroem o hipergrafo representado pela Figura 2.111 (p. 92)(a). Todavia, quando executamos o Algoritmo 2.85 para construir L a partir da remoção sucessiva das arestas incidentes em vértices de grau 1, ficamos com o hipergrafo da Figura 2.111(d), em que ainda sobram arestas mas faltam vértices de grau 1. A execução do Algoritmo 2.85, portanto, falha, e precisamos fazer mais um sorteio. Finalmente, suponhamos que no próximo sorteio conseguimos:

$$\begin{pmatrix} & h_0 \colon U \to \{0,1,2\} & h_1 \colon U \to \{3,4,5\} & h_2 \colon U \to \{6,7,8\} \\ \texttt{gott} & 0 & 3 & 6 \\ \texttt{fried} & 0 & 3 & 8 \\ \texttt{wil} & 1 & 3 & 7 \\ \texttt{helm} & 2 & 5 & 8 \\ \texttt{von} & 2 & 4 & 8 \\ \texttt{lei} & 0 & 4 & 6 \\ \texttt{bniz} & 0 & 3 & 7 \end{pmatrix} \tag{2.109}$$

Removendo sucessivamente as arestas incidentes sobre vértices de grau 1, conseguimos ficar com nenhuma aresta e construir a lista

$$L = [\{1,3,7\},\{2,5,8\},\{2,4,8\},\{0,4,6\},\{0,3,6\},\{0,3,7\},\{0,3,8\}], \tag{2.110}$$

como ilustramos na Figura 2.112 (p. 93). Consequentemente, concluímos a fase de mapeamento.

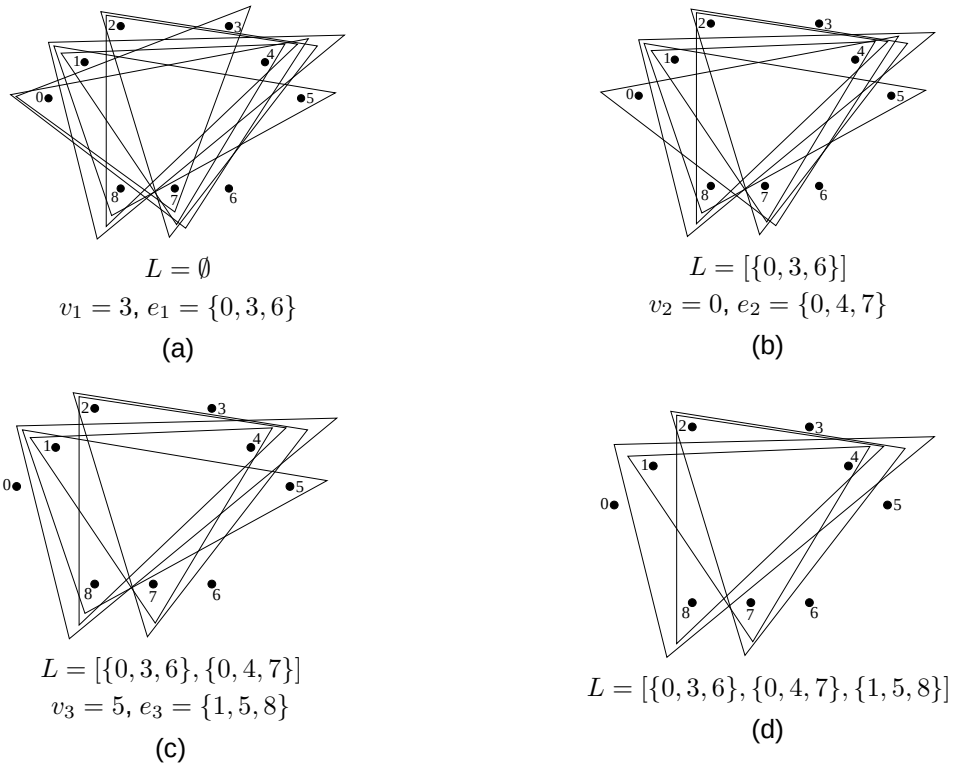


FIGURA 2.111 — Um exemplo da fase de mapeamento do 3-BDZ.

Embora ainda não tenhamos definido como se dá o sorteio das r funções h_j, é claro que gostaríamos que todas elas pudessem ser sorteadas em tempo total $O(rn) = O(n)$. Na Seção 2.4.4, quando discutirmos as diferentes abordagens para esse sorteio, notaremos que nenhuma delas custa tempo $\omega(n)$.

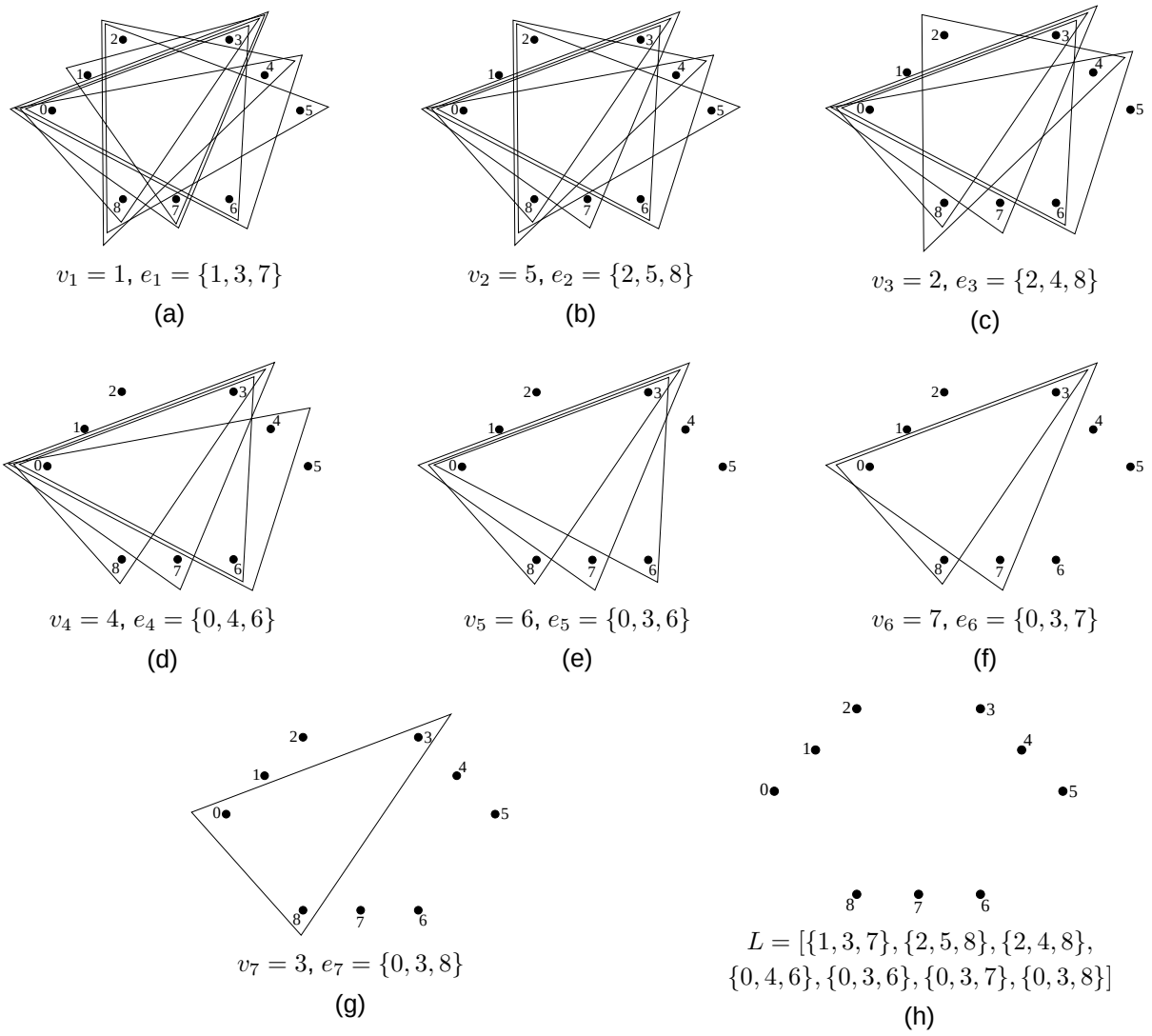


FIGURA 2.112 — Outro exemplo da fase de mapeamento do 3-BDZ.

Verificar se existem x e y distintos tais que $e(x) = e(y)$ também pode ser feito em tempo $O(n)$ se utilizamos, por exemplo, alguma estrutura de dados auxiliar que armazena as imagens de S por cada h_j e aponta quando ocorre de algum y marcar na estrutura toda uma sequência de $(h_0(y), \ldots, h_{r-1}(y))$ já marcada. Também não é difícil perceber que o Algoritmo 2.85 custa tempo $O(n)$ se previamente já temos os graus dos vértices e as arestas incidentes a cada vértice, informações que podem ser determinadas na própria construção do hipergrafo. Enfim, como, do Teorema 2.92, o

número esperado de iterações é constante, temos que o tempo esperado de toda a fase de mapeamento do r-BDZ é $O(n)$.

2.4.2 Fase de designação

Na fase de mapeamento, conseguimos ordenar as arestas de um r-hipergrafo r-partido G_r numa lista $L = [e_1, \ldots, e_n]$ de modo que toda aresta e_j é incidente num vértice v_j tal que $v_j \notin e_{j'}$ para todo $j' > j$. Logo, se varremos L começando por e_n e terminando em e_1, visitando todos os vértices das arestas, em cada aresta podemos encontrar ao menos um vértice ainda não visitado. Assim, podemos designar cada aresta e_j por um de seus vértices — qualquer um daqueles que ainda não tenham sido visitados no instante em que a varredura de L visita e_j. Sendo $V_0, \ldots, V_{r-1}$ as partes de $V(G)$, queremos, então, para cada aresta $e = \{u_0, \ldots, u_{r-1}\}, u_0 \in V_0, \ldots, u_{r-1} \in V_{r-1}$, encontrar o índice $i \in \{0, \ldots, r-1\}$ que identifique o vértice u_i que designará a aresta e.

O índice i do vértice u_i que designará a aresta $\{u_0, \ldots, u_{r-1}\}$ pode ser calculado como uma soma módulo r de rótulos g atribuídos a todos os vértices daquela aresta. Assim, a fase de designação do r-BDZ consiste em encontrar uma rotulação dos vértices apropriada $g\colon V(G) \to \{0, \ldots, r\}$ que faça com que seja injetiva a função

$$\rho\colon E(G_r) \to V(G_r) \qquad \text{definida por} \qquad \rho(\{u_0, \ldots, u_{r-1}\}) = u_{i(\{u_0,\ldots,u_{r-1}\})}, \tag{2.113}$$

sendo $i(\{u_0, \ldots, u_{r-1}\}) \in \{0, \ldots, r-1\}$ o índice

$$i(\{u_0, \ldots, u_{r-1}\}) = \big(g(u_0) + \cdots + g(u_{r-1})\big) \bmod r \tag{2.114}$$

do vértice que designa a aresta $\{u_0, \ldots, u_{r-1}\}$. Observe-se que, por ser ρ injetiva, a *hash*

$$p\colon U \to \{0, \ldots, t-1\} \qquad \text{definida por} \qquad p(x) = \rho(e(x)) \tag{2.115}$$

por si só já seria uma *hash* perfeita para S, embora não mínima.

Quando a varredura da lista L se encontra numa aresta e_j, já foram visitados aqueles e somente aqueles vértices que pertencem a cada $e_{j'}, j' > j$. Então, para rotularmos um vértice u_i ainda não visitado e fazermos com que

$$\left(\sum_{v \in e_j} g(v) \right) \bmod r = i, \tag{2.116}$$

com o intuito de designar e_j por u_i, basta que atribuamos a u_i o rótulo

$$g(u_i) = \left(i - \sum\nolimits_{v \in e_j \text{ já visitado}} g(v) \right) \bmod r, \tag{2.117}$$

tendo sido inicializados todos os vértices com o rótulo r, como mostramos a seguir. Atente-se para que os demais vértices em e_j que ainda não tenham sido visitados simplesmente continuarão com o rótulo r, o que implica que um vértice fica com rótulo diferente de r se e só se designa uma aresta.

TEOREMA 2.118: *Após a execução do Algoritmo 2.120 (p. 96) para um hipergrafo G_r e uma lista $L = [e_1, \ldots, e_n]$ apropriada, tem-se uma injeção ρ de $E(G_r)$ em $V(G_r)$, sendo que um vértice u de G_r é imagem de alguma aresta por ρ se e só se $g(u) \neq r$.*

Demonstração: Seja ρ como definida na Equação 2.113 (p. 94). Na primeira iteração da varredura de L, nenhum vértice de $e_n = \{u_{0_{(n)}}, \ldots, u_{r-1_{(n)}}\}$ foi ainda visitado. Um deles, $u_{i_{(n)}}$, será escolhido na linha 6 e receberá o rótulo $g(u_{i_{(n)}}) = i_{(n)}$, sendo marcado como visitado. Os rótulos de todos os outros vértices de e_n, que também serão marca-

dos como visitados, continuarão sendo iguais a r e não serão alterados em nenhuma outra iteração do laço da linha 5, já que apenas vértices não visitados têm seus rótulos alterados (linhas 6 e 7). Assim,

$$i(e_n) = \big(i_{(n)} + (r-1)r\big) \bmod r = i_{(n)}, \tag{2.119}$$

e, consequentemente, $u_{i_{(n)}}$ é a imagem de e_n por ρ. Não obstante, $g(u_{i_{(n)}}) \neq r$, e $g(u_{i'}) = r$ para todo $u_{i'} \in e_n$ que não é $u_{i_{(n)}}$. Ademais, a função parcial ρ_n, definida apenas pela imagem de e_n por ρ, é injetiva.

ALGORITMO 2.120 — Fase de designação do r-BDZ.

ENTRADA: um hipergrafo G_r e uma ordenação $L = [e_1, \ldots, e_n]$ das arestas de G_r tal que toda aresta e_j é incidente a um vértice v_j tal que $v_j \notin e_{j'}$ para todo $j' > j$.

SAÍDA: uma rotulação g dos vértices que torne injetiva a função ρ definida pela Equação 2.113 (p. 94).

1: PARA CADA $u \in V(G_r)$, FAÇA:
2: $g(u) \leftarrow r$;
3: `visitado`$(u) \leftarrow$ NÃO,
4: FIM.
5: PARA CADA j DE n ATÉ 1, FAÇA:
6: PARA ALGUM $u_i \in e_j = \{u_0, \ldots, u_{r-1}\}$ ainda não visitado, FAÇA:
7: $g(u_i) \leftarrow \big(i - \sum_{v \in e_j \text{ já visitado}} g(v)\big) \bmod r$;
8: `visitado`$(u_i) \leftarrow$ SIM;
9: PARA CADA $u_{i'} \in e_j, u_{i'} \neq u_i$ ainda não visitado, FAÇA:
10: `visitado`$(u_{i'}) \leftarrow$ SIM,
11: FIM,
12: FIM,
13: FIM.

Suponhamos por indução que, para todo j de n até 2 valha que, na iteração $e_j = \{u_{0_{(j)}}, \ldots, u_{r-1_{(j)}}\}$ do laço da linha 5:

a) $g(u_{i_{(j)}}) = i_{(j)} \neq r$, sendo $u_{i_{(j)}}$ o vértice escolhido na linha 6;

b) $g(u_{i'}) = r$ para todo não visitado $u_{i'} \in e_j$ que não é $u_{i_{(j)}}$;

c) a função parcial ρ_j, definida apenas pelas imagens de $e_j, \ldots, e_n$ por ρ, é injetiva.

Considerando agora a iteração $e_{j-1} = \{u_{0_{(j-1)}}, \ldots, u_{r-1_{(j-1)}}\}$ do laço da linha 5, tomemos $u_{i_{(j-1)}}$ o vértice escolhido na linha 6, ao qual se atribui o rótulo

$$g\big(u_{i_{(j-1)}}\big) = \left(i_{(j-1)} - \sum\nolimits_{v \in e_{j-1} \text{ já visitado}} g(v)\right). \tag{2.121}$$

Ora, como $u_{i_{(j-1)}}$ não fora ainda visitado, $u_{i_{(j-1)}} \notin e_{j'}$ para todo $j' > j-1$, tal como os rótulos dos outros vértices não visitados de e_{j-1} continuarão iguais a r. Assim,

$$\begin{aligned} i(e_{j-1}) &= \left(g\big(u_{i_{(j-1)}}\big) + \sum\nolimits_{v \in e_{j-1} \text{ já visitado}} g(v) + \sum\nolimits_{v \in e_{j-1} \text{ ainda não visitado}} r\right) \bmod r \\ &= \left(i_{(j-1)} + \sum\nolimits_{v \in e_{j-1} \text{ ainda não visitado}} r\right) \bmod r \\ &= i_{(j-1)}, \end{aligned} \tag{2.122}$$

e, consequentemente, $u_{i_{(j-1)}}$ é a imagem de e_{j-1} por ρ. Não obstante, $g(u_{i_{(j-1)}}) \neq r$ e $g(u_{i'}) = r$ para todo não visitado $u_{i'} \in e_{j-1}$ que não é $u_{i_{(j-1)}}$. Ademais, a função ρ_{j-1}, parcial se $j > 2$ e total caso contrário, definida apenas pelas imagens de $e_{j-1}, \ldots, e_n$ por ρ, é injetiva. Note-se que, se $j = 2$, $\rho_{j-1} = \rho_1 = \rho$. □

Exemplificamos no Quadro 2.123 a execução da fase da designação do 3-BDZ para o hipergrafo G_3 e a lista L apresentados na Figura 2.112 (p. 93).

e_j	u_i escolhido	$g(u_i)$	$i(e_j)$	$\rho(e_j)$
$e_7 = \{0,3,8\}$	$u_0 = 0$	$(0-(3+3)) \bmod 3 = 0$	$(0+3+3) \bmod 3 = 0$	0
$e_6 = \{0,3,7\}$	$u_2 = 7$	$(2-(0+3)) \bmod 3 = 2$	$(0+3+2) \bmod 3 = 2$	7
$e_5 = \{0,3,6\}$	$u_2 = 6$	$(2-(0+3)) \bmod 3 = 2$	$(0+3+2) \bmod 3 = 2$	6
$e_4 = \{0,4,6\}$	$u_1 = 4$	$(1-(0+2)) \bmod 3 = 2$	$(0+2+2) \bmod 3 = 1$	4
$e_3 = \{2,4,8\}$	$u_0 = 2$	$(0-(2+3)) \bmod 3 = 1$	$(1+2+3) \bmod 3 = 0$	2
$e_2 = \{2,5,8\}$	$u_1 = 5$	$(1-(1+3)) \bmod 3 = 0$	$(1+0+3) \bmod 3 = 1$	5
$e_1 = \{1,3,7\}$	$u_0 = 1$	$(0-(3+2)) \bmod 3 = 1$	$(1+3+2) \bmod 3 = 0$	1

QUADRO 2.123 — Um exemplo da fase de designação do 3-BDZ.

Podemos verificar que a fase de mapeamento do r-BDZ custa tempo $O(n)$, já

que o laço da linha 1 do Algoritmo 2.120 é executado $|V(G_r)| = c(r)n$ vezes e o laço da linha 5, cujas iterações custam tempo $O(r) = O(1)$ cada, é executado n vezes.

2.4.3 Fase de *ranking*

Na fase de designação, conseguimos uma injeção ρ de $E(G_r)$ em $V(G_r) = \{0, \ldots, t-1\}$, $t = c(r)n$, o que implica numa *hash* $p\colon x \to \rho(e(x))$ perfeita, mas não mínima, para S. Na fase de *ranking*, estabelecemos uma função $\mathrm{rank}\colon V(G_r) \to \{0, \ldots, n-1\}$, que mapeia injetivamente $\rho(E(G_r))$ — o conjunto dos vértices que designam arestas — para $\{0, \ldots, n-1\}$, o que nos traz uma *hash*

$$\wp\colon U \to \{0, \ldots, n-1\}, \qquad \text{definida por} \qquad \wp(x) = \mathrm{rank}(\rho(e(x))), \tag{2.124}$$

perfeita e mínima para S.

TEOREMA 2.125: *A função* $\mathrm{rank}\colon V(G_r) \to \{0, \ldots, n-1\}$*, definida por*

$$\mathrm{rank}(u) = |\{v \in V(G_r) : v < u \text{ e } g(v) \neq r\}|, \tag{2.126}$$

é injetiva sobre $\rho(E(G_r))$*.*

Demonstração: Sejam e_j e $e_{j'}$ duas arestas distintas de G_r tais que $\rho(e_j) = u$ e $\rho(e_{j'}) = u'$. Como, do Teorema 2.118 (p. 95), ρ é injetiva, suponhamos, sem perda de generalidade, que $u < u'$. Portanto, todo vértice $v < u$ tal que $g(v) \neq r$ é também menor que u'. Ainda do Teorema 2.118, ambos $g(u)$ e $g(u')$ são diferentes de r. Assim,

$u \in \{v \in V(G_r) : v < u'$ e $g(v) \neq r\}$, e, consequentemente,

$$\{u\} \cup \{v \in V(G_r) : v < u \text{ e } g(v) \neq r\} \subseteq \{v \in V(G_r) : v < u' \text{ e } g(v) \neq r\}. \quad (2.127)$$

Uma vez que, evidentemente, $u \notin \{v \in V(G_r) : v < u$ e $g(v) \neq r\}$,

$$\begin{gathered} |\{u\} \cup \{v \in V(G_r) : v < u \text{ e } g(v) \neq r\}| = \operatorname{rank}(u) + 1 \leqslant \\ |\{v \in V(G_r) : v < u' \text{ e } g(v) \neq r\}| = \operatorname{rank}(u'). \end{gathered} \quad (2.128)$$

Dessarte, $\operatorname{rank}(u) < \operatorname{rank}(u')$, e rank é injetiva sobre $\rho(E(G_r))$. □

A fase de *ranking* do r-BDZ consiste meramente na construção duma estrutura de dados que permita a computação de $\operatorname{rank}(u)$ em tempo $O(1)$ para todo vértice u. Como a função g é computável em $O(1)$, isso pode ser feito através duma simples varredura sobre $V(G_r)$ (Algoritmo 2.129), de tempo $O(t) = O(n)$, cuja execução para a função g apresentada no Quadro 2.123 (p. 97) exibimos no Quadro 2.130 (p. 100). Exibimos ainda, no Quadro 2.131 (p. 100), a composição das funções das três fases do 3-BDZ para o conjunto de chaves $S = \{\texttt{gott}, \texttt{fried}, \texttt{wil}, \texttt{helm}, \texttt{von}, \texttt{lei}, \texttt{bniz}\}$.

ALGORITMO 2.129 — Fase de *ranking* do r-BDZ.

ENTRADA: uma função $g\colon V(G_r) \to \{0, \ldots, r\}$ que torna ρ (Equação 2.113, p. 94) injetiva.
SAÍDA: uma função $\operatorname{rank}\colon V(G_r) \to \{0, \ldots, n-1\}$ injetiva sobre $\rho(E(G_r))$.

```
1: Σ ← 0.
2: PARA CADA u DE 0 ATÉ t − 1, FAÇA:
3:     rank(u) ← Σ;
4:     SE g(u) ≠ r, ENTÃO,
5:         Σ ← Σ + 1,
6:     FIM,
7: FIM.
```

u	$g(u)$	$\{v \in V(G_r) : v < u$ e $g(v) \neq 3\}$	$\mathrm{rank}(u)$
0	0	$\emptyset$	0
1	1	$\{0\}$	1
2	1	$\{0,1\}$	2
3	3	$\{0,1,2\}$	3
4	2	$\{0,1,2\}$	3
5	0	$\{0,1,2,4\}$	4
6	2	$\{0,1,2,4,5\}$	5
7	2	$\{0,1,2,4,5,6\}$	6
8	3	$\{0,1,2,4,5,6,7\}$	7

QUADRO 2.130 — Um exemplo da fase de *ranking* do 3-BDZ.

x	$e(x)$	$\rho(e(x))$	$\wp(x) = \mathrm{rank}(\rho(e(x)))$
`gott`	$\{0,3,6\}$	6	5
`fried`	$\{0,3,8\}$	0	0
`wil`	$\{1,3,7\}$	1	1
`helm`	$\{2,5,8\}$	5	4
`von`	$\{2,4,8\}$	2	2
`lei`	$\{0,4,6\}$	4	3
`bniz`	$\{0,3,7\}$	7	6

QUADRO 2.131 — Um exemplo do 3-BDZ.

Temos dito, então, que a complexidade de tempo do r-BDZ é $O(n)$. A seguir, trataremos da análise do espaço das *hashes* produzidas pelo esquema.

2.4.4 Análise de espaço

Uma *hash* $\wp$ construída pelo r-BDZ é representada pelas funções $h_0, \ldots, h_{r-1}$, g e rank, e por nada mais. A rigor, cada h_j seria sorteada do espaço de todas as funções de U em $\left\{\left\lfloor \frac{jt}{r} \right\rfloor, \ldots, \left\lfloor \frac{(j+1)t}{r} \right\rfloor - 1\right\}$ com distribuição uniforme. Isso não é necessário, já que estamos interessados apenas na imagem de h_j por S. Ainda assim, entretanto, se de fato sorteássemos cada h_j do espaço de todas as funções de S em

$\left\{\left\lfloor\frac{jt}{r}\right\rfloor,\ldots,\left\lfloor\frac{(j+1)t}{r}\right\rfloor-1\right\}$ com distribuição uniforme, precisaríamos de cerca de $n\lg\frac{t}{r}$ *bits* para representar cada h_j. Dessarte, $\wp$ requereria no mínimo $nr\lg\frac{t}{r}=O(n\log n)$ *bits*, e não conseguiríamos uma *hash* perfeita e mínima sobre S representável por $O(n)$ *bits*, como prometido.

Botelho, Pagh e Ziviani (2007) sugerem várias abordagens pseudoaleatórias para reduzir o espaço da representação das h_j para o esperado, como a estratégia de *dividir e compartilhar — split and share —* de Dietzfelbinger e Weidling (2007). Em todas essas abordagens, requer-se um número menor de *bits* verdadeiramente aleatórios no sorteio de cada h_j, sendo os demais *gerados* a partir desses *bits* de modo que o conjunto de todos os *bits* se comporte como se fosse integralmente aleatório. Com isso, para armazenar cada h_j precisamos apenas daqueles *bits* que foram realmente sorteados, já que os demais são oriundos do processo pseudoaleatório que constituiu h_j.

Quando da implementação, os autores dizem construir as h_j a partir de uma *hash* h', que mapeia cada chave x para uma cadeia $h'(x)=(h'(x)[0],\ldots,h'(x)[\gamma-1])$ de γ *bits*, sendo γ uma constante. Se β é a constante escolhida que define o número de *bits* de $h'(x)$ que devem ser usados para construir cada h_j, e se $h'(x)[a..b]$ denota o número natural cuja representação binária é a subcadeia de $h'(x)$ que começa na posição a e termina na posição b, então, cada função h_j pode ser dada por

$$h_j(x)=\big(h'(x)[j\beta..(j+1)\beta-1]\big)\bmod\Big(\frac{m}{r}\Big)+j\Big(\frac{m}{r}\Big). \tag{2.132}$$

A função h' é escolhida de uma família de *hashes* universais proposta no trabalho de Alon et al. (1999). Sendo L um limitante superior para o tamanho das representações binárias das chaves, $h'(x)=\mathrm{A}\boldsymbol{x}^{\mathrm{T}}$, sendo A uma matriz $\gamma\times L$ preenchida com *bits* aleatórios e $\boldsymbol{x}$ a representação binária de x. Dessa forma, h', assim como toda função h_j, é computável em tempo constante para qualquer x. Ademais, o espaço

necessário para se representarem todas as h_j é apenas o espaço necessário para se representar A, um número constante de $\gamma \cdot L$ *bits*. Botelho, Pagh e Ziviani (2007) ainda sugerem a abordagem de Alon e Naor (1996) para que h' seja implementada eficientemente.

Porque o rank de um vértice é um número em $\{0, \ldots, n-1\}$, armazenar cada $\mathrm{rank}(u)$ custa $\lfloor \lg(n-1) \rfloor + 1$ *bits*. Se optássemos por representar a função rank por uma estrutura de dados que armazena $\mathrm{rank}(u)$ para todo u, precisaríamos de $\Omega(n \log n)$ *bits*, e a *hash* $\wp$ não seria representável por $O(n)$ *bits*, como desejado. Inspirados no clássico trabalho de Pagh (2002), os autores do BDZ mostram que não precisamos armazenar todos os valores de rank para mantermos a função computável em tempo $O(1)$. Sendo ζ uma constante real positiva pré-determinada, e sendo $\kappa = \lfloor \frac{\lg t}{\zeta} \rfloor$, podemos optar por armazenar numa estrutura de dados $\mathrm{rankTable}$ apenas os valores $\mathrm{rank}(0)$, $\mathrm{rank}(\kappa)$, $\mathrm{rank}(2\kappa)$ etc. Para tanto, acompanhamos a $\mathrm{rankTable}$ de uma estrutura auxiliar com t *bits* que, para todo vértice u que não está na $\mathrm{rankTable}$, permite contar em tempo $O(\frac{1}{\zeta})$ quantos vértices entre v e $u-1$ têm rótulo diferente de r, a fim de calcular $\mathrm{rank}(u)$, sendo v o maior vértice na $\mathrm{rankTable}$ menor que u. Botelho, Pagh e Ziviani (2007) também afirmam que tanto $\mathrm{rankTable}$ quanto a estrutura auxiliar podem juntas ser representadas por $\zeta \cdot t$ *bits*.

Como o contradomínio de g é o conjunto $\{0, \ldots, r\}$, armazenar g requer cerca de $t \lg r$ *bits*. Somando o espaço para se representarem as h_j com o espaço para rank e g, temos que o espaço necessário para $\wp$ é cerca de

$$\gamma \cdot L + \zeta \cdot t + t \lg r = O(t) = O(n) \tag{2.133}$$

bits, como prometido.

No caso particular em que $r = 2$, temos do Teorema 2.92 (p. 87) que $t = (2+\epsilon)n$. Como cada vértice é rotulado com um valor g em $\{0, 1, 2\}$, representar g

requer 2 *bits* por vértice. Logo, o espaço total para se representar $\wp$ é de cerca de

$$\gamma \cdot L + \zeta \cdot t + 2t = \gamma \cdot L + (2 + \zeta)(2 + \epsilon)n \tag{2.134}$$

bits. Como convencionamos $\epsilon = 0{,}09$ e $\zeta = 0{,}125$, e supondo que n é suficientemente grande para que a constante $\gamma \cdot L$, quando dividida por n, se aproxime de 0, temos que uma *hash* construída pelo 2-BDZ requer apenas aproximadamente 4.44 *bits* por chave.

O esquema para o qual $r = 3$ chamamos simplesmente de BDZ porque é o que traz os melhores resultados. Como $c(3) = 1{,}23$, e como cada rótulo, elemento de $\{0, 1, 2, 3\}$, também só precisa de 2 *bits*, o espaço total da representação da *hash* construída é de cerca de

$$\gamma \cdot L + \zeta \cdot t + 2t = \gamma \cdot L + (2 + \zeta)(1{,}23)n \tag{2.135}$$

bits. Como convencionamos $\zeta = 0{,}125$, e também assumindo que $\frac{\gamma \cdot L}{n} \approx 0$, temos que uma *hash* construída pela BDZ requer apenas aproximadamente $2{,}62$ *bits* por chave, como anunciado.

Analogamente, no caso genérico em que $r \geqslant 2$, o tamanho da representação de uma *hash* construída pelo r-BDZ é aproximadamente $(2 + \zeta)(c(r))$ *bits* por chave. Por exemplo, se $\zeta = 0{,}125$, esse tamanho é de aproximadamente $2{,}77$ *bits* por chave no 4-BDZ e de aproximadamente $3{,}04$ *bits* por chave no 5-BDZ, de acordo com o Quadro 2.105 (p. 90).

2.5 O ESQUEMA J-BDZ

Na seção anterior, apresentamos o esquema de *hashing* BDZ, que constrói em tempo $O(n)$ para um conjunto com n chaves uma *hash* mínima e perfeita representável por aproximadamente $\gamma \cdot L + 2{,}62n$ *bits*. Assim, o BDZ, também chamado de 3-BDZ, é um esquema mínimo, perfeito, prático, ótimo em tempo e próximo de ótimo em espaço. Além dos $2{,}62$ *bits* por chave, os $\gamma \cdot L$ *bits* devem-se à representação da matriz A, com γ linhas e L colunas, utilizada para a computação das *hashes* h_0, h_1 e h_2, que mapeiam as chaves para arestas de um 3-hipergrafo 3-partido. Lembre-se que L é um limitante superior para os tamanhos das chaves e γ é uma constante arbitrária, que determina, para toda chave x, o tamanho das cadeias de *bits* $h'(x)$, da qual tiramos as subcadeias $h_0(x)$, $h_1(x)$ e $h_2(x)$.

Como explicado na Seção 2.4.4 (p. 101), o sorteio das funções h_0, h_1 e h_2 na fase de mapeamento do BDZ é dado na realidade através do sorteio dos *bits* da matriz A. Embora $\gamma \cdot L$ seja uma constante, ainda assim o número de *bits* que precisam ser sorteados é considerável, mesmo que o número esperado de iterações da fase de mapeamento seja apenas 1. Se L, por exemplo, é 256, no caso em que as chaves estão codificadas segundo a tabela ASCII, e se γ é 32, no caso em que as cadeias $h'(x)$ têm o tamanho de uma palavra de um processador de 32 *bits*, o número de *bits* sorteados para preencher a matriz A é de 8192. Assim, a menos que o número de chaves seja maior que 8192, não teremos $2{,}62$ *bits* por chave na representação da *hash*, mas sim no mínimo $3{,}62$.

Outra desvantagem de se sortearem as funções h_0, h_1 e h_2 através da matriz A é que não levamos em conta que, na prática, frequentemente as chaves gozam de muita similitude, distanciando-se muito da distribuição uniforme idealizada pela teoria. Ademais, embora a computação de $\{h_0(x), h_1(x), h_2(x)\}$ para uma chave x possa

ser realizada em tempo $O(1)$, a constante dessa computação é consideravelmente elevada: são $O(\gamma \cdot L)$ instruções, que precisam ser executadas não apenas durante a construção da *hash* h pelo BDZ, mas sempre que for computado o endereço $h(x)$ associado a x.

Assim como usamos as práticas *hashes* de Jenkins (1997), J_1, J_2 e J_3, apresentadas no Apêndice B, para o sorteio de h_1 e h_2 no J-BMZ, podemos convenientemente proceder de igual modo para o sorteio de h_0, h_1 e h_2 na fase de mapeamento do 3-BDZ, quando tratamos especificamente do caso em que as chaves são cadeias de caracteres, como *URLs*. Dessarte, usamos J_1 para h_0, J_2 para h_1 e J_3 para h_2, como exibimos no Algoritmo 2.136.

ALGORITMO 2.136 — Fase de mapeamento do J-BDZ.

ENTRADA: um conjunto S com n chaves.

SAÍDA: um 3-hipergrafo 3-partido G_3 e uma ordenação $L = \{e_1, \ldots, e_n\}$ das arestas tal que, para todo $j \in \{1, \ldots, n\}$, existe um $v_j \in e_j$ tal que $v_j \notin e_{j'}$ para todo $j' > j$.

1: $t \leftarrow 1{,}23n$.
2: $V(G) \leftarrow \{1, \ldots, t\}$.
3: Sorteie uma semente s de 32 *bits*.
4: PARA CADA $x \in S$, FAÇA:
5: $\quad h_0(x) \leftarrow J_1(x, s) \bmod t$;
6: $\quad h_1(x) \leftarrow J_2(x, s) \bmod t$;
7: $\quad h_2(x) \leftarrow J_3(x, s) \bmod t$,
8: FIM.
9: SE existem em S distintos x e y tais que $h_j(x) = h_j(y)$ para todo $j \in \{0, \ldots, r-1\}$, ENTÃO,
10: $\quad$ RETORNE à linha 3,
11: FIM.
12: $E(G) \leftarrow \{\{h_0(x), \ldots, h_{r-1}(x)\} : x \in S\}$.
13: $G \leftarrow (V(G), E(G))$.
14: SE a execução do Algoritmo 2.85 (p. 84) para G devolve ERRO, ENTÃO,
15: $\quad$ RETORNE à linha 3;
16: SENÃO, SE devolve $L = \{e_1, \ldots, e_n\}$, ENTÃO,
17: $\quad$ DEVOLVA G e L,
18: FIM.

O uso das *hashes* de Jenkins é vantajoso porque cada $J_i, i \in \{1, 2, 3\}$, distribui bem as chaves entre as 2^{32} valores na tabela *hash* de J_i, mesmo que as chaves sejam

dotadas de muita similitude, dado que cada *bit* de x interfere em todos os *bits* do endereço associado a x pela *hash* J_i. Também é muito eficiente a computação das *hashes*, as quais podem ser computadas em paralelo. No código proposto pelo autor, cada *hash* requer a execução apenas de $6|x| + 35$ instruções.

A implementação do BDZ na biblioteca CMPH utiliza as *hashes* de Jenkins J_1, J_2 e J_3 para a fase de mapeamento, basicamente como apresentamos no Algoritmo 2.136 (p. 105). Essa versão do BDZ decidimos chamar de *J-BDZ*.

No J-BDZ, as fases de designação e *ranking* procedem exatamente como descritas respectivamente nos Algoritmos 2.120 e 2.129 para $r = 3$. Note-se que uma *hash* construída pelo J-BDZ requer apenas $32 + 2{,}62n$ *bits* para ser representada, pois a representação das *hashes* h_0, h_1 e h_2 requer apenas a representação da semente s de 32 *bits* utilizada.

3 METODOLOGIA E RESULTADOS

Em 2009, o Prof. Dr. Jair Donadelli Jr., então professor adjunto da Universidade Federal do Paraná (UFPR), recomendou ao seu orientando de mestrado, Leandro M. Zatesko, que estudasse o trabalho de Botelho, Kohayakawa e Ziviani (2005), com o objetivo de construir uma demonstração para a conjectura que os autores deixaram (Conjectura 2.64, p. 69; Teorema 3.1, p. 109). Além disso, o professor ainda recomendou ao seu aluno que tentasse desaleatorizar o algoritmo BMZ, dados os estudos que o aluno já fizera sobre Pseudoaleatoridade e Semialeatoridade de Grafos. A 30 de setembro de 2010, os professores Dr. Jair Donadelli Jr., Dr. André Luiz P. Guedes e Dr. André Luís Vignatti aprovaram a proposta do mestrando, que expôs à banca os seguintes objetivos para seu trabalho:

a) propor um esquema *hashing* perfeito, mínimo, prático e determinístico, resultante da desaleatorização do BMZ (objetivo primário);
b) fornecer uma fundamentação teórica satisfatória para o esquema proposto (objetivo primário);
c) completar as demonstrações apresentadas na fundamentação teórica do BMZ, provando ser verdadeira a conjectura;
d) implementar o algoritmo determinístico proposto e apresentar resultados empíricos, comparando-os com os resultados do algoritmo original, aleatorizado.

Consideramos atingidos todos os objetivos propostos, havendo feito na realidade muito mais do que aquilo a que nos propusemos. Enquanto estudávamos o esquema BMZ, descobrimos a CMPH e o esquema BDZ. Assim, resolvemos estender nossa estratégia de desaleatorização não apenas para o BMZ, mas também para o BDZ. Para sermos justos em nossa apresentação de resultados empíricos, resolve-

mos implementar nossos esquemas determinísticos a partir da implementação que os próprios autores das versões aleatorizadas já haviam desenvolvido na CMPH, fazendo simples alterações no código original. Ademais, executamos ambas as versões — a determinística e a aleatorizada — de cada algoritmo numa mesma máquina com as mesmas entradas. Por fim, quando da escrita deste texto, ainda conseguimos chegar a uma demonstração da Conjectura de Botelho, Kohayakawa e Ziviani (2005), objetivo que já abandonáramos. Começaremos a apresentação de nossos resultados a partir da demonstração que propomos para essa conjectura. Depois, nas Seções 3.2 (p. 111), 3.3 (p. 117) e 3.4 (p. 122), exporemos nossos algoritmos determinísticos, mostrando os resultados que obtivemos empiricamente.

3.1 Demonstração da Conjectura de Botelho, Kohayakawa e Ziviani (2005)

Conforme estabelecemos na Notação 2.63 (p. 69), usamos N_{t} para denotar a soma total de reassociações que a fase de busca do BMZ faz para rotular todos os vértices críticos de um grafo G (Algoritmo 2.58, p. 67). A parte crítica da fase de busca do BMZ nada mais é que uma busca em largura no subgrafo crítico de G, munida de uma estratégia gulosa de rotulação. Assim, usamos N_{bedges} para denotar o número de arestas críticas de G que figuram como arestas de retorno nessa busca. Botelho, Kohayakawa e Ziviani (2005) demonstraram, como transcrevemos no Teorema 2.66 (p. 69), que, se vale que $N_{\mathsf{t}} \leqslant N_{\mathsf{bedges}}$, o rótulo máximo atribuído a algum vértice segundo essa estratégia gulosa é $n - 1$, o que assegura mínima a *hash* construída pelo BMZ.

Contudo, por não conseguirem mostrar que $N_{\mathsf{t}} \leqslant N_{\mathsf{bedges}}$, os autores exibiram a proposição como uma conjectura (Conjectura 2.64, p. 69). Ao mostrarmos, no Teo-

rema 3.1, que a conjectura de fato era verdadeira, como Botelho, Kohayakawa e Ziviani (2005) já haviam percebido empiricamente, não apenas apresentamos um resultado até então não provado, mas também encerramos a demonstração da corretude do BMZ.

Construímos nossa demonstração através da sobreposição de duas provas por indução. Na primeira, tentamos estabelecer uma injeção do conjunto das reassociações no conjunto das arestas de retorno. Para tanto, utilizamos uma segunda prova por indução em que mostramos que cada reassociação incluída no passo da primeira indução é mapeado para uma aresta de retorno até então não utilizada.

TEOREMA 3.1: $N_{\mathrm{t}} \leqslant N_{\mathsf{bedges}}$.

Demonstração: Seja B um conjunto que começa vazio na execução do Algoritmo 2.58 (p. 67). É trivial que a rotulação do vértice inicial da busca u_0 não causa nenhuma reassociação. Vamos mostrar por indução que toda reassociação acrescenta a B uma aresta de retorno que não pertencia a B. Do Lema A.29 (p. 140), todo vértice $v \neq u_0$ que a linha 4 desenfileira para que seja rotulado na linha 14 possui um único vizinho v_0, evidentemente já rotulado, que enfileirou v. Se tentar rotular v com um valor i causa uma reassociação r, então, há algum w vizinho de v tal que $g(w)+i = j = g(u_1)+g(u_2)$ para alguma aresta já rotulada $\{u_1, u_2\}$ tal que $g(u_1) < g(u_2)$.

Se r é a primeira reassociação, e se $w = v_0$, então, $\{w, v\}$ é uma aresta da árvore de busca e, por conseguinte, do Lema A.36 (p. 142), $\{u_1, u_2\}$ é uma aresta de retorno. Nesse caso, já que B começa vazio, basta fazermos $B \leftarrow B \cup \{u_1, u_2\}$. Por outro lado, se r é a primeira reassociação, mas se $w \neq v_0$, então, $\{w, v\}$ é uma aresta de retorno, pois, não fosse, teríamos um ciclo na árvore da busca, já que, por w e v_0 já estarem rotulados na ocasião da rotulação de v, haveria na árvore um caminho entre w e v_0 que não passa por v. Nesse caso, basta fazermos $B \leftarrow B \cup \{w, v\}$ que

estaremos adicionando a B uma aresta de retorno que antes não lhe pertencia.

Se r não é a primeira reassociação, suponhamos por hipótese de indução que toda reassociação r' anterior a r satisfaz a propriedade de haver acrescentado a B uma aresta de retorno que não pertencia a B antes de r'.

Se $w = v_0$, então, a aresta $\{w, v\}$ é uma aresta da árvore de busca e, por conseguinte, do Lema A.36 (p. 142), $f_0 = \{u_1, u_2\}$ é uma aresta de retorno. Notemos que, quando da ocasião da tentativa de se rotular v com i, se f_0, rotulada com j, já está em B, então, existe uma outra aresta f_1 rotulada depois de f_0 mas antes de $\{w, v\}$ que fez com que f_0 entrasse em B porque tentamos rotular f_1 com $h(f_0) = j$. Se f_1 também já está em B quando da ocasião da tentativa de se rotular v com i, então, analogamente, existe uma outra aresta f_2 rotulada depois de f_1 mas antes de $\{w, v\}$ que fez com que f_1 entrasse em B porque tentamos rotular f_2 com $h(f_1) > j$. Assim, indutivamente, existe algum $k \geqslant 0$ para o qual f_k não está em B. Ademais, a aresta f_k, por ser rotulada com um valor no mínimo j, e por causa do Lema A.36 (p. 142), seguramente é uma aresta de retorno. Façamos, então, $B \leftarrow B \cup \{f_k\}$.

Se, por outro lado, $w \neq v_0$, a aresta $\{w, v\}$ é uma aresta de retorno. Na ocasião da tentativa de se rotular v com i, $\{w, v\}$ só pode estar em B se houve alguma tentativa anterior de se rotular v com algum $i' < i$ que pôs $f_0 = \{w, v\}$ em B por $j' = g(w) + i'$ já ser um rótulo atribuído a alguma outra aresta f_1. Por sua vez, f_1 só pode estar em B se existe uma outra aresta f_2 rotulada depois de f_1 mas antes de f_0 que fez com que f_1 entrasse em B porque tentamos rotular f_2 com $h(f_1) = j'$. Se f_2 também já está em B quando da ocasião da tentativa de se rotular v com i', então, analogamente, existe uma outra aresta f_3 rotulada depois de f_2 mas antes de f_0 que fez com que f_2 entrasse em B porque tentamos rotular f_3 com $h(f_2) > j'$. Assim, indutivamente, existe algum $k \geqslant 0$ para o qual f_k não está em B. Ademais, a aresta f_k, por ser rotulada com um valor no mínimo j', e por causa do Lema A.36, seguramente é uma aresta de retorno. Façamos, então, $B \leftarrow B \cup \{f_k\}$.

Mostramos que toda reassociação executada pelo Algoritmo 2.58 acrescenta a B uma aresta de retorno que não estava lá antes. Ao final da execução do algoritmo, temos uma injeção do conjunto de todas as reassociações para o conjunto das arestas de retorno. Portanto, $N_{\mathsf{t}} \leqslant N_{\mathsf{bedges}}$. □

3.2 O esquema D-BMZ

Na primeira das fases do BMZ, a fase de mapeamento, mapeamos um conjunto S com n chaves para o conjunto das arestas de um grafo com $1{,}15n$ vértices através do sorteio de duas funções h_1 e h_2, sendo que cada chave x é mapeada para a aresta $\{h_1(x), h_2(x)\}$. O sorteio das funções h_1 e h_2, por sua vez, dá-se através do sorteio de $2L|\Sigma|$ números em $\{0, \ldots, t-1\}$ para preencher duas tabelas, T_1 e T_2, que determinam h_1 e h_2 como exibido na Equação 2.29 (p. 51).

Um par (T_1, T_2) bem pode ser visto como um número $T = (z_{2L|\Sigma|-1} \ldots z_0)_t$ de $2L|\Sigma|$ dígitos na base t, se concatenamos todas as linhas das duas tabelas, vindo T_1 por primeiro, e lemos o resultado da concatenação de trás para frente, apenas para facilitar o processo. Assim, $T_1[1,1]$ é o dígito z_0 de T, $T_1[1,2]$ é o $z_1 \ldots$ $T_1[2,1]$ é o $z_{|\Sigma|} \ldots$ $T_2[1,1]$ é o $z_{L|\Sigma|} \ldots$ Mais formalmente, o dígito z_j, $0 \leqslant j < 2L|\Sigma|$, de T na base t é

dado por $T_a[b, c]$, sendo:

$$
\begin{aligned}
a &= \begin{cases} 1, & \text{se } j < L|\Sigma|; \\ 2, & \text{caso contrário;} \end{cases} \\
b &= \begin{cases} \left\lfloor \frac{j}{|\Sigma|} \right\rfloor + 1, & \text{se } j < L|\Sigma|; \\ \left\lfloor \frac{j}{|\Sigma|} \right\rfloor + 1 - L, & \text{caso contrário;} \end{cases} \\
c &= (j \bmod |\Sigma|) + 1.
\end{aligned}
\tag{3.2}
$$

Agora, se um par (T_1, T_2) pode ser visto como um número T de $2L|\Sigma|$ dígitos na base t, então, sortear (T_1, T_2) bem pode ser entendido como sortear um número T em $\{0, \ldots, N\}$, sendo $N = \big((t-1)(t-1)\ldots(t-1)\big)_t$ o maior número que pode ser representado por $2L|\Sigma|$ dígitos na base t:

$$
N = (t-1)t^0 + \cdots + (t-1)t^{2L|\Sigma|-1} = t^{2L|\Sigma|} - 1. \tag{3.3}
$$

Conforme argumentamos na Seção 2.2.1 (p. 54), a probabilidade de um par (T_1, T_2) servir para a construção de G é aproximadamente $\frac{1}{2{,}13} \cong 0{,}469$. Dessarte, aproximadamente $0{,}469$ dos números T em $\{0, \ldots, N\}$ servem para a fase de mapeamento do BMZ. Evidentemente, se um número T não serve, a probabilidade de $T-1$ ou $T+1$ não servirem é alta, já que possivelmente a alteração da apenas uma célula nas tabelas T_1 e T_2 não promove significativa mudança na estrutura do grafo. Assim, podemos estabelecer um procedimento determinístico para a fase de mapeamento do BMZ a partir duma ordenação $T^{(0)}, \ldots, T^{(N)}$ de todos os números em $\{0, \ldots, N\}$. Iniciamos o procedimento pegando o primeiro $T = T^{(0)}$; se não servir, pegamos o próximo $T = T^{(1)}$, e assim por diante.

Precisamos apenas garantir que, repetindo o procedimento N vezes, todos os números em $\{0, \ldots, N\}$ são contemplados. Ainda, se construímos a ordenação

de tal modo que as tabelas T_1 e T_2 associadas a $T^{(j)}$, para todo $j \in \{0, \ldots, N-1\}$, são bem diferentes das tabelas associadas a $T^{(j+1)}$, podemos esperar um pequeno número de iterações desse procedimento, talvez até um pouco menor do que o número de iterações do procedimento aleatório, já que seguramente não caímos no risco de repetir um par (T_1, T_2). É claro, que, no pior caso, quando todos os números que não servem são os $(1 - 0{,}469)$ primeiros da ordenação e todos os que servem são os $0{,}469$ últimos, precisamos fazer aproximadamente $(1 - 0{,}469)(N+1) \cong 0{,}53t^{2L|\Sigma|} = O(t^{2L|\Sigma|}) = O(n^{2L|\Sigma|})$ iterações, o que resulta num indesejável tempo $O(n^{2L|\Sigma|+1})$ para a fase de mapeamento. No entanto, esperamos no caso médio que o número de iterações seja próximo de $2{,}13$, como tínhamos na versão aleatorizada. Há de se lembrar também que, na versão aleatorizada, o algoritmo nem possuía garantia de parada, já que os sorteios eram feitos com reposição. Agora, na versão determinística, temos garantido que jamais ressorteamos um mesmo par (h_1, h_2).

Mostramos no Teorema 3.4 um modo de se construir uma ordenação σ a partir de $\sigma_0 = 0$ para que $\{\sigma_0, \ldots, \sigma_N\} = \{0, \ldots, N\}$.

TEOREMA 3.4: *Se $t-1$ é divisível por 3, se $\sigma_0 = 0$, e se*

$$\sigma_{j+1} = \left(\sigma_j + \frac{N}{3}\right) \bmod (N+1) \qquad \textit{(3.5)}$$

para todo $j \in \{0, \ldots, N-1\}$, então, $\{\sigma_0, \ldots, \sigma_N\} = \{0, \ldots, N\}$.

Demonstração: Se $t-1$ é divisível por 3, então,

$$N = (t-1)^0 + (t-1)^1 + (t-1)^2 + \cdots + (t-1)^{2L|\Sigma|} \qquad (3.6)$$

também o é, e $\frac{N}{3}$ é um inteiro cujos fatores primos são os mesmos de N, à possível

exceção do fator 3, caso $\frac{N}{3}$ não seja divisível por 3. Portanto, como nenhum dos primos que divide N divide $N+1$, temos que $\frac{N}{3}$ e $N+1$ são primos entre si.

Já que $\sigma_0 = 0$, temos que $\sigma_j = \left(j\frac{N}{3}\right) \bmod (N+1)$ para todo $j \in \{0, \ldots, N\}$. Mas como $N+1$ e $\frac{N}{3}$ são coprimos, $N+1$ nunca divide $j\frac{N}{3}$ para qualquer $j \in \{1, \ldots, N\}$. Assim, $\sigma_j \neq 0$ para todo $j \neq 0$.

Suponhamos por contradição que exista ao menos uma repetição em σ e tomemos σ_j o primeiro elemento repetido, igual a $\sigma_{j'}$ para algum $j' < j$. Ora, como $j > 0$, $\sigma_j \neq 0$, $\sigma_{j'} \neq 0$, e, consequentemente, $j' \neq 0$. Contudo,

$$\left((j - j')\frac{N}{3}\right) \bmod (N+1) = 0, \tag{3.7}$$

e, por conseguinte, $\sigma_{j-j'} = 0$, um absurdo, já que $j - j' > 0$. □

Note-se que somar um σ_j com $\frac{N}{3}$ módulo $N+1$ é simplesmente somar, propagando *carry* e descartando o *overflow*, cada célula das tabelas de σ_j com $\frac{t-1}{3}$, já que a representação da $\frac{N}{3}$ na base t possui todos os dígitos iguais a $\frac{t-1}{3}$. Evidentemente, $\sigma_0 = 0$ nunca é apropriado, já que faz com que h_1 e h_2 mapeiem todas as chaves para o vértice 0. Então, convencionamos começar nosso procedimento por algum outro número $T^{(0)} \in \{0, \ldots, N\}$. Escolhemos o número

$$T^{(0)} = (\underbrace{\ldots, 1, 0, t-1, t-2, \ldots, 2, 1, 0}_{2L|\Sigma| \text{ dígitos}})_t, \tag{3.8}$$

e definimos indutivamente, para $j \in \{1, \ldots, N\}$,

$$T^{(j)} = \left(T^{(j-1)} + \frac{N}{3}\right) \bmod (N+1). \tag{3.9}$$

Como é de se supor, escolhemos 3 para dividir N porque $3 > 2{,}13$, para que números

similares ocorram na ordenação apenas de 3 em 3 iterações.

Apelidamos de D-BMZ a versão determinística do BMZ cuja fase de mapeamento apresentamos no Algoritmo 3.10 (p. 115). Deve-se ressaltar que o D-BMZ difere do BMZ apenas na fase de mapeamento, prosseguindo de modo idêntico nas outras fases. Vale frisar também que não necessariamente fazemos $t = 1{,}15n$. Se $1{,}15n - 1$ não for divisível por 3, fazemos, outrossim, $t - 1$ ser o primeiro número divisível por 3 depois de $1{,}15n - 1$.

ALGORITMO 3.10 — Fase de mapeamento do D-BMZ.

ENTRADA: um conjunto S de n chaves.
SAÍDA: um grafo G com t vértices e n arestas, sendo t o primeiro número no mínimo $1{,}15n$ tal que $t-1$ é divisível por 3.
1: Faça $T \leftarrow (z_{2L|\Sigma|-1} \dots z_0)_t$, sendo $z_j = (t-1) - (j \bmod t)$, e, assim, tenha T_1 e T_2.
2: PARA CADA $x \in S$, FAÇA:
3: calcule $h_1(x)$ e $h_2(x)$ pela Equação 2.29 (p. 51);
4: SE $\{h_1(x), h_2(x)\} = \{h_1(y), h_2(y)\}$ para algum outro $y \in S$, ENTÃO,
5: faça $T \leftarrow \left(T + \frac{N}{3}\right) \bmod (N+1)$, e, assim, tenha outras tabelas T_1 e T_2;
6: reinicie o laço da linha 2,
7: FIM,
8: FIM.
9: DEVOLVA $(\{0, \dots, t-1\}, \{\{h_1(x), h_2(x)\} : x \in S\})$.

Concluímos a fundamentação teórica do D-BMZ no Teorema 3.13 (p. 116), cujo enunciado requer a notação que introduzimos a seguir.

NOTAÇÃO 3.11: Sendo S um conjunto com n chaves, definimos a sequência binária $B(S) = (b_0 \dots b_N)$, na qual:

$$b_j = \begin{cases} 1, & \text{se } T^{(j)} \text{ serve para } S; \\ 0, & \text{caso contrário.} \end{cases} \tag{3.12}$$

Mostramos no Teorema 3.13 que, no caso médio, a fase de mapeamento do D-BMZ é iterada aproximadamente $2{,}13$ vezes, considerando-se todos os possíveis

conjuntos de chaves S equiprováveis. Para tanto, assumimos uma hipotése de que o valor esperado de X é no máximo o valor esperado de Y, sendo X a variável aleatória que conta o número de iterações para um conjunto S tomado com distribuição uniforme, e sendo Y a variável aleatória que informa o primeiro índice j de uma sequência $(b_0 \dots b_N)$ tal que b_j é o primeiro *bit* não-nulo da sequência, tomada com distribuição também uniforme do espaço de todas as sequências binárias com exatos $p(N+1)$ *bits* iguais a 1, sendo $p = \mathrm{e}^{-\frac{1}{(1,15)^2}} \cong 0{,}469 \cong \frac{1}{2{,}13}$. Consideramos essa hipótese razoável de se assumir, já que cada conjunto S induz uma sequência $B(S)$, e já que acreditamos que as piores sequências $(b_0 \dots b_N)$ são na realidade as que têm menor probabilidade de serem induzidas por algum S.

TEOREMA 3.13: *Sendo Y uma variável aleatória que informa o primeiro índice j tal que b_j é o primeiro bit igual a 1 numa sequência binária $(b_0 \dots b_N)$ tomada com distribuição uniforme do espaço de todas as sequências com exatos $p(N+1)$ bits iguais a 1, $\mathbb{E}Y \leqslant \frac{1}{p}$.*

Demonstração: Das propriedades da esperança,

$$\mathbb{E}Y = \sum_{j=0}^{N} \mathbb{E}(Y|e_j)\mathbb{P}(e_j) = \sum_{j=0}^{N} j\mathbb{P}(e_j), \tag{3.14}$$

sendo e_j o evento em que $b_0 = \cdots = b_{j-1} = 0$ e $b_j = 1$. Como

$$\mathbb{P}(e_j) = \begin{cases} p, & \text{se } j = 0, \text{ e} \\ (1-p)^{j-1}p, & \text{caso contrário,} \end{cases} \tag{3.15}$$

temos que

$$\mathbb{E}Y = p\sum_{j=0}^{N} j(1-p)^{j-1}. \tag{3.16}$$

Mas, do Lema A.38 (p. 143), $\sum_{j=0}^{N} j(1-p)^{j-1} \leqslant \frac{1}{(1-(1-p))^2}$. Logo, $\mathbb{E}Y \leqslant \frac{1}{p}$, como queríamos mostrar. □

Exemplificamos a construção das tabelas T_1 e T_2 da fase de mapeamento do D-BMZ num hipotético caso em que temos $L = |\Sigma| = 3$ e $t = 7$. Note-se que precisamos somar cada célula de cada tabela com 2, propagando *carry* e descartando o possível *overflow* final. As quatro primeiras iterações do D-BMZ resultariam nas seguintes tabelas, ignorando-se o sucesso ou o fracasso de cada iteração:

$$
\begin{aligned}
T = 321065432106543210 \therefore T_1 = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 0 & 1 \end{bmatrix}, T_2 &= \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 0 \\ 1 & 2 & 3 \end{bmatrix}; && (1^{\text{a}} \text{ iteração}) \\
T = 543320654332065432 \therefore T_1 = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 0 \\ 2 & 3 & 3 \end{bmatrix}, T_2 &= \begin{bmatrix} 4 & 5 & 6 \\ 0 & 2 & 3 \\ 3 & 4 & 5 \end{bmatrix}; && (2^{\text{a}} \text{ iteração}) \\
T = 065543206554320654 \therefore T_1 = \begin{bmatrix} 4 & 5 & 6 \\ 0 & 2 & 3 \\ 4 & 5 & 5 \end{bmatrix}, T_2 &= \begin{bmatrix} 6 & 0 & 2 \\ 3 & 4 & 5 \\ 5 & 6 & 0 \end{bmatrix}; && (3^{\text{a}} \text{ iteração}) \\
T = 321065432106543206 \therefore T_1 = \begin{bmatrix} 6 & 0 & 2 \\ 3 & 4 & 5 \\ 6 & 0 & 1 \end{bmatrix}, T_2 &= \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 0 \\ 1 & 2 & 3 \end{bmatrix}. && (4^{\text{a}} \text{ iteração})
\end{aligned}
\tag{3.17}
$$

3.3 O esquema D-BDZ

É fácil perceber que a mesma estratégia de desaleatorização usada para o BMZ pode ser usada para o BDZ. Basta que estabeleçamos um processo determinístico que itere sobre todas as possíveis matrizes $\mathsf{A}_{\gamma \times L}$. Uma matriz A com γ linhas e L colunas bem pode ser vista como um número A de $\gamma \cdot L$ dígitos na base 2. Dessarte, sortear A pode ser entendido como sortear um número A em $\{0, \ldots, N\}$, sendo, evi-

dentemente,

$$N = 2^{\gamma \cdot L} - 1. \tag{3.18}$$

Na Seção 2.4.4 (p. 101), assumindo que sortear uma matriz A com distribuição uniforme é praticamente o mesmo que sortear um r-hipergrafo r-partido G_r com distribuição suficientemente próxima da uniforme, argumentamos no Teorema 2.92 (p. 87) que a probabilidade de um sorteio de uma matriz A servir — ou de um sorteio de um número A em $\{0, \ldots, N\}$ servir — é aproximadamente 1. Entretanto, ocorrendo a improbabilidade de um sorteio de um A não servir, a improbabilidade de $A - 1$ ou $A + 1$ também não servirem é mais considerável que a improbabilidade de um A' mais distante de A não servir. Estabelecemos dessa forma um procedimento determinístico para a fase de mapeamento do BDZ também a partir de uma ordenação $A^{(0)}, \ldots, A^{(N)}$ de todos os números em $\{0, \ldots, N\}$. Iniciamos o procedimento com $A = A^{(0)}$; se não servir, tentamos o próximo $A = A^{(1)}$, e assim por diante.

Precisamos garantir que, repetindo o procedimento supracitado N vezes, todos os números em $\{0, \ldots, N\}$ serão contemplados. Dos $N+1$ números em $\{0, \ldots, N\}$, $(1 - p)(N + 1)$ servem, para algum $p \approx 0$, conforme a Equação 2.103 (p. 89). Então, no pior caso, precisamos iterar a fase de mapeamento do BDZ $p(N + 1) + 1 = O(1)$ vezes. Então, mesmo no pior caso, nossa versão determinística do BDZ faz apenas $O(1)$ iterações na fase de mapeamento, não alterando o tempo $O(n)$ de toda a fase de mapeamento. Há de se lembrar que, na versão aleatorizada, o algoritmo nem possuía garantia de parada, já que os sorteios eram feitos com reposição. Garantindo na versão determinística que jamais ressorteamos uma mesma matriz A, reduzimos o número de iterações da fase de mapeamento no pior caso de $+\infty$ para $O(1)$.

Mostramos a seguir um modo de se construir uma ordenação σ a partir de um $\sigma_0 = 0$ para que $\{\sigma_0, \ldots, \sigma_N\} = \{0, \ldots, N\}$.

TEOREMA 3.19: *Se $\gamma \cdot L$ é par, então, N é divisível por 3. Ademais, se $\sigma_0 = 0$, e se*

$$\sigma_{j+1} = \left(\sigma_j + \frac{N}{3}\right) \bmod (N+1) \tag{3.20}$$

para todo $j \in \{0, \ldots, N-1\}$, então, $\{\sigma_0, \ldots, \sigma_N\} = \{0, \ldots, N\}$.

Demonstração: Se $\gamma \cdot L$ é par, então,

$$\begin{aligned} N &= \sum_{j=0}^{\gamma \cdot L - 1} 2^j \\ &= \sum_{\substack{j=0 \\ j \text{ par}}}^{\gamma \cdot L - 1} 2^j + \sum_{\substack{j=0 \\ j \text{ ímpar}}}^{\gamma \cdot L - 1} 2^j \\ &= \sum_{\substack{j=0 \\ j \text{ par}}}^{\gamma \cdot L - 1} 2^j + \sum_{\substack{j=0 \\ j \text{ par}}}^{\gamma \cdot L - 1} 2 \cdot 2^j \\ &= 3 \sum_{\substack{j=0 \\ j \text{ par}}}^{\gamma \cdot L - 1} 2^j, \end{aligned} \tag{3.21}$$

e, consequentemente,

$$\sum_{\substack{j=0 \\ j \text{ par}}}^{\gamma \cdot L - 1} 2^j = \frac{N}{3}. \tag{3.22}$$

Logo, N é divisível por 3. Ademais, nenhum dos primos que divide N divide $N+1$, o que nos traz que $\frac{N}{3}$ e $N+1$ são coprimos.

Já que $\sigma_0 = 0$, temos que $\sigma_j = \left(j\frac{N}{3}\right) \bmod (N+1)$ para todo $j \in \{0, \ldots, N\}$. Mas como $N+1$ e $\frac{N}{3}$ são coprimos, $N+1$ nunca divide $j\frac{N}{3}$ para qualquer $j \in \{1, \ldots, N\}$. Assim, $\sigma_j \neq 0$ para todo $j \neq 0$.

Seguindo a demonstração como no Teorema 3.4, suponhamos por contradição que exista ao menos uma repetição em σ e tomemos σ_j o primeiro elemento repetido, igual a $\sigma_{j'}$ para algum $j' < j$. Ora, como $j > 0$, $\sigma_j \neq 0$, $\sigma_{j'} \neq 0$, e, consequentemente,

$j' \neq 0$. Contudo,

$$(j - j')\frac{N}{3} \bmod (N+1) = 0, \tag{3.23}$$

e, por conseguinte, $\sigma_{j-j'} = 0$, um absurdo, já que $j - j' > 0$. □

Da Equação 3.22, a representação binária de $\frac{N}{3}$ é a cadeia $(01\ldots0101)_2$. Assim, somar um σ_j com $\frac{N}{3}$ módulo $N+1$ é simplesmente somar, propagando *carry* e descartando *overflow*, cada z_j de $A = (z_{\gamma\cdot L-1}\ldots z_0)$ com 1 se j é par ou com 0 caso contrário. É claro que $\sigma_0 = 0$ nunca é apropriado, já que faz com que h_0, h_1 e h_2 mapeiem as chaves todas para a mesma aresta. Então, convencionamos começar nosso procedimento por algum outro número $A^{(0)} \in \{0,\ldots,N\}$. Escolhemos o número $A^{(0)} = (10011001\ldots)_2$, e definimos indutivamente para $j \in \{1,\ldots,N\}$,

$$A^{(j)} = \left(A^{(j-1)} + \frac{N}{3}\right) \bmod (N+1). \tag{3.24}$$

Nossa versão determinística do BDZ apelidamos de D-BDZ, que difere da versão aleatorizada apenas no sorteio das funções h_0, h_1 e h_2 na fase de mapeamento. Nas demais fases, o BDZ e o D-BDZ são idênticos. Deve-se apenas ter o cuidado de escolher um γ apropriado para que $\gamma \cdot L$ seja par.

A Notação 3.25, a qual introduzimos a seguir, é necessária para o enunciado do Teorema 3.27, que conclui a fundamentação do D-BDZ.

DEFINIÇÃO 3.25: Sendo S um conjunto com n chaves, definimos a sequência binária $B(S) = (b_0\ldots b_N)$, na qual:

$$b_j = \begin{cases} 1, & \text{se } A^{(j)} \text{ serve para } S; \\ 0, & \text{caso contrário.} \end{cases} \tag{3.26}$$

O Teorema 3.27 enuncia que, no caso médio, a fase de mapeamento do D-BDZ é iterada aproximadamente 1 vez apenas, considerando-se todos os possíveis conjuntos de chaves S equiprováveis. Para tanto, assumimos uma hipotése de que o valor esperado de X é no máximo o valor esperado de Y, sendo X a variável aleatória que conta o número de iterações para um conjunto S tomado com distribuição uniforme, e sendo Y a variável aleatória que informa o primeiro índice j de uma sequência $(b_0 \dots b_N)$ tal que b_j é o primeiro *bit* não-nulo da sequência, tomada com distribuição também uniforme do espaço de todas as sequências binárias com exatos $p(N+1)$ *bits* iguais a 0, sendo $p \approx 0$. Consideramos essa hipótese razoável de se assumir, já que cada conjunto S induz uma sequência $B(S)$, e já que acreditamos que as piores sequências $(b_0 \dots b_N)$ são na realidade as que têm menor probabilidade de serem induzidas por algum S.

TEOREMA 3.27: *Sendo Y uma variável aleatória que informa o primeiro índice j tal que b_j é o primeiro bit igual a 1 numa sequência binária $(b_0 \dots b_N)$ tomada com distribuição uniforme do espaço de todas as sequências com exatos $p(N+1)$ bits iguais a 0, sendo $p \approx 0$, $\mathbb{E}Y \approx 0$.*

Demonstração: Das propriedades da esperança,

$$\mathbb{E}Y = \sum_{j=0}^{N} \mathbb{E}(Y|e_j)\mathbb{P}(e_j) = \sum_{j=0}^{N} j\mathbb{P}(e_j), \tag{3.28}$$

sendo j o evento em que $b_0 = \cdots = b_{j-1} = 0$ e $b_j = 1$. Como

$$\mathbb{P}(e_j) = \begin{cases} 1-p, & \text{se } j = 0, \text{ e} \\ p^{1-j}(1-p), & \text{caso contrário,} \end{cases} \tag{3.29}$$

temos que

$$\mathbb{P}(e_j) \approx \begin{cases} 1, & \text{se } j = 0; \\ 0, & \text{caso contrário.} \end{cases} \tag{3.30}$$

Dessarte, $\mathbb{E}Y \approx 0$, como queríamos mostrar. □

Num hipotético caso em que $\gamma = 4$ e $L = 3$, as quatro primeiras iterações do D-BDZ resultariam nas seguintes matrizes, ignorando-se o sucesso ou o fracasso de cada iteração:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}; \quad A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}; \quad A = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}. \tag{3.31}$$

3.4 Os esquemas D-J-BMZ e D-J-BDZ

Vimos nas Seções 2.3 e 2.5 que o sorteio das funções da fase de mapeamento tanto do J-BMZ quanto do J-BDZ se dá através das computações das *hashes* de Jenkins para uma semente aleatória s de 32 *bits*. Como cada *bit* da semente aleatória interfere em todos os *bits* de qualquer endereço computado por uma *hash* de Jenkins, não faz muita diferença a ordem em que pegamos os 2^{32} possíveis valores para s. Para obedecermos o padrão que vimos adotado, tomamos a primeira semente s como um valor determinado (0 não parece ser uma boa ideia, como sempre) e, cada vez que s não serve, somamo-lo módulo 2^{32} com um coprimo de 2^{32}. Evidentemente, essa soma custa apenas uma instrução de um processador de 32 *bits*, já que se trata de uma soma com *overflow* simples de duas palavras do processador. Determinamos a

semente inicial como sendo $s = \frac{n}{3}$, e escolhemos $\frac{2^{32}-1}{3} = 1431655765$ o coprimo de 2^{32} a ser somado a cada iteração.

Para podermos empírica mas efetivamente comparar nossos algoritmos determinísticos com suas versões aleatorizadas, resolvemos apenas adaptar o código original, como implementado na CMPH, que os próprios autores dos esquemas desenvolveram para apresentarem seus resultados em seus artigos. Como já comentado, os algoritmos que constam na biblioteca são o J-BMZ e o J-BDZ, além de outros esquemas de *hashing* mínimos e perfeitos. Portanto, os esquemas que implementamos foram o D-J-BMZ e o D-J-BDZ, nossas variantes determinísticas para, respectivamente, os esquemas J-BMZ e J-BDZ.

Executamos todos os esquemas na mesma máquina: um AMD Athlon™ 3500+ com 64 kiB de cache L1, 512 kiB de cache L2 e 1 GiB de memória RAM. Não queríamos que os resultados fossem muito influenciados pela sorte. Como, evidentemente, rodar o mesmo algoritmo várias vezes para a mesma entrada pode ser interessante para algoritmos aleatorizados, mas inútil para algoritmos determinísticos, decidimos rodar cada caso de teste para 50 instâncias diferentes e, ao final, tomar a média aritmética dos 50 resultados obtidos. Contudo, num mesmo caso de teste utilizamos o mesmo conjunto de 50 instâncias para todos os algoritmos.

Apresentamos nossos resultados no Quadro 3.32. Comparamos não apenas o tempo gasto pelos algoritmos, mas também o número de iterações da fase de mapeamento de cada um deles, o qual pode ser identificado no quadro como uma unidade a mais do número de ressorteios de tuplas de funções. Perceba-se que os resultados obtidos são bastante próximos, o que nos atesta que, como já supúnhamos, nossas variantes determinísticas são empiricamente equivalentes aos algoritmos aleatorizados originais.

As bases de dados de nossos casos de teste são *URLs* artificiais, geradas por um *script* que assegura produzir bases de dados com n URLs todas distintas,

$n =$	$6\,250\,000$		$12\,500\,000$		$25\,000\,000$	
esquema	ressorteios	tempo (s)	ressorteios	tempo (s)	ressorteios	tempo (s)
J-BMZ	$0{,}8800$	$30{,}4516$	$1{,}4400$	$70{,}6006$	$1{,}4200$	$151{,}2864$
D-J-BMZ	$1{,}3600$	$32{,}9100$	$0{,}9000$	$65{,}1682$	$0{,}8800$	$144{,}6608$
J-BDZ	0	$23{,}2462$	0	$47{,}8154$	0	$101{,}2546$
D-J-BDZ	0	$23{,}0818$	0	$49{,}0736$	0	$102{,}4540$

QUADRO 3.32 — Resultados empíricos.

recebendo n como entrada. Na implementação de qualquer esquema na CMPH, se fornecemos a qualquer um dos algoritmos uma entrada com URLs repetidas, a fase de mapeamento jamais consegue obter sucesso. Para garantir que as n *URLs* produzidas fossem todas distintas, nosso *script* adiciona à j-ésima *URL* a marca j antes do ponto que antecede o *TLD* (*Top-Level Domain*) da *URL*. Por exemplo, uma execução do *script* para $n = 20$ produziu:

```
http://no.yqsk0.uvf.fh/
http://ja.sacwkcfmouuwqslpuaxr1.lxh.gf/
http://zh.çhçvçm2.wns/z7ob.cey
http://ru.zqcyeçrniqmfvlsocbf3.gcn/9j.itz
http://www.ncbcmrçoxksurifwwlusqsdasawsxxjçp4.wve/
http://pt.cewdpbrz5.bço.bj/sj1o4.jer
https://fi.qfxlxlzo6.ift/kaoir2.hnç
http://www.vgxjmçqdkkmyjncnxdnmdnqphquçjggnu7.sgv/9rnl_=b.syç
http://www.qqgvdyazqbljnfhl8.sls/3t_hlb=c7.hun
http://ro.xliaeczmawrjpçtuhçvh9.ull.kt/_zvs6lu8l2.xav
http://www.engbcemavpwqpm10.lod/
http://sk.pçainp11.tcq/
https://www.npwmxtvvyhqltpnyhvgojdnvçhdyaiks12.wgh/4ma?qkd6.xjy
http://it.pmmaçkndeez13.gyn/
```

http://www.dysçmubkylz14.dfx.ij/m.bjh

http://www.vwlajphkyarem15.lou/

http://www.bçzl16.xçl.az/

http://www.bgksttwxcpunoggvçptyo17.cfd.ix/5juçj.beh

http://www.tddyztgjoideaçchjdestlloçgççm18.kic.av/

http://dccksvrltçmçqtqkv19.exl.qh/

http://sr.jmemfwyeylgairkrzroouiusnvsltdjm20.fhd/

4 CONCLUSÃO E TRABALHOS FUTUROS

Mapear n objetos para $\{0, \ldots, n-1\}$ de modo que não haja dois objetos distintos mapeados para o mesmo número e não fique número algum em $\{0, \ldots, n-1\}$ sem ser utilizado no mapeamento é um problema muito comum em Ciência da Computação. Como defendido por Botelho, Kohayakawa e Ziviani (2005), deparamo-nos com ele em esquemas de verificação e autenticação, em sistemas de impressões digitais, em códigos de correção de erros, em funções de criptografia, em estruturas de dados e em aplicações de armazenamento eficiente em memória e recuperação rápida em conjuntos estáticos, como palavras reservadas em linguagens de programação ou sistemas interativos, *URLs* em sistemas de busca da Internet, ou conjuntos de itens em técnicas de mineração de dados. Encontrar essas bijeções é uma das aplicações da Teoria de *Hashing*, já que cada bijeção pode ser considerada como uma *hash* perfeita e mínima para o conjunto S dos n objetos, subconjunto dum universo U. Um algoritmo que constrói uma *hash* perfeita e mínima para S que mapeia o universo U para $\{0, \ldots, n-1\}$ chamamos de esquema de *hashing*.

Como definimos no Capítulo 1, um esquema de *hashing*:

a) é dito perfeito quando sempre constrói para S uma *hash* que não mapeia duas chaves distintas para o mesmo endereço;
b) é dito mínimo quando sempre constrói para S uma *hash* pela qual não ficam em $\{0, \ldots, n-1\}$ endereços sem ser associados a chaves de S;
c) é dito eficiente em tempo quando é executável em tempo $O(n \log n)$;
d) é dito ótimo em tempo quando é executável em tempo $O(n)$;
e) é dito eficiente em espaço quando o tamanho da representação de qualquer *hash* que construa, em *bits*, é $O(n)$;
f) é dito ótimo em espaço quando o tamanho da representação de qualquer

hash que construa, em *bits*, é $n \lg e + \lg \lg u + O(\log n)$;

g) é dito prático quando seus resultados empíricos para casos de teste viáveis atestam seus resultados teóricos;

h) é dito determinístico quando, para um mesmo conjunto de chaves S, constrói sempre a mesma *hash*;

i) é dito aleatorizado quando, para um mesmo conjunto de chaves S, pode construir diferentes *hashes* a cada execução.

No Capítulo 2, apresentamos os esquemas de *hashing*

a) FKS, de Fredman, Komlòs e Szemerédi (1984),

b) BMZ, de Botelho, Kohayakawa e Ziviani (2005), e

c) BDZ, de Botelho, Pagh e Ziviani (2007).

Escolhemos apresentar o FKS por se tratar dum esquema clássico, útil para ser comparado com os outros. O BMZ apresentamos por ser o esquema que motivou o presente trabalho. Enquanto do estudo do BMZ, encontramos o BDZ, um esquema melhor, e resolvemos estender nosso estudo também para ele. O BMZ também pode ser chamado de BKZ, assim como o BDZ pode ser batizado de BPZ.

Construções de *hashes* podem derivar de resultados de diversas disciplinas da Matemática. O FKS, por exemplo, é um esquema construído basicamente a partir de demonstrações de Teoria dos Números: encadeamos *hashes* intermediárias, que são somas modulares, para que, no final, tenhamos todas as colisões resolvidas. O BMZ, por sua vez, é um esquema que mapeia as chaves para as arestas de um grafo para, então, com a rotulação dos vértices do grafo obter a rotulação das arestas. Já o BDZ é um esquema que mapeia as chaves para as arestas de um 3-hipergrafo 3-partido para, então, com a rotulação dos vértices do hipergrafo obter a rotulação das arestas. O BDZ é na realidade o r-BDZ quando $r = 3$. Poderíamos utilizar qualquer outro $r \geqslant 2$. No entanto, os melhores resultados são obtidos quando $r = 3$. Precisamos advertir o leitor de que fomos nós quem batizamos os esquemas da família

do BDZ de r-BDZ, para evitarmos ambiguidade.

Tanto o BMZ quanto o BDZ podem usar as *hashes* de Jenkins, propostas por Jenkins (1997), em suas fases de mapeamento, nas quais as chaves são mapeadas para arestas de um grafo — no caso do BMZ — ou de um 3-hipergrafo — no caso do BDZ. Essa abordagem possui várias vantagens, pois reduz o tamanho da representação das *hashes* construída, diminui o tempo da computação dos mapeamentos e atesta bons resultados para os casos reais de aplicação de *Hashing*, nos quais as chaves frequentemente possuem alguma similitude. Nós batizamos essas variantes de J-BMZ e de J-BDZ. É importante frisar que são esses os algoritmos implementados pelos autores na biblioteca CMPH.

Em cada iteração da fase de mapeamento do BMZ, mapeamos o conjunto de chaves S para um conjunto de pares de vértices $\{\{h_1(x), h_2(x)\} : x \in S\}$ através do sorteio de duas tabelas T_1 e T_2 que geram as funções h_1 e h_2. Se esse sorteio não serve, por não gerar um grafo sem laços e sem arestas múltiplas, simplesmente ressorteamos as tabelas T_1 e T_2. Os autores mostraram que a probabilidade de um par (T_1, T_2) servir é aproximadamente $0{,}469$. Assim, o número esperado de iterações para a fase de mapeamento é de aproximadamente $2{,}13$.

Como cerca de $0{,}469$ de todos os pares (T_1, T_2) servem para a fase de mapeamento, simplesmente propusemos uma ordenação de todos os pares possíveis de tal modo que pares consecutivos fossem significativamente diferentes. Assim, conseguimos estabelecer um processo determinístico para a fase de mapeamento do BMZ. Sempre tentamos o primeiro par de tabelas (T_1, T_2) da ordenação estabelecida. Se não funciona, pegamos o próximo par na ordenação, e, assim sucessivamente, até conseguirmos um par de tabelas que sirva. Se os autores mostraram que o tempo esperado da fase de mapeamento do BMZ é $O(n)$, nós mostramos que $O(n)$ é igualmente o tempo do caso médio da fase de mapeamento de nossa variante determinística, sendo também $2{,}13$ o número esperado de iterações.

Estendemos essa simples estratégia de desaleatorização para o BDZ, o J-BMZ e o J-BDZ. Batizamos nossas variantes determinísticas de D-BMZ, D-BDZ, D-J-BMZ e D-J-BDZ. Comparamos todos esses esquemas de *hashing* no Quadro 4.1. Como o leitor há de notar, os resultados das versões aleatorizadas são praticamente equivalentes aos das versões determinísticas. Com isso, mostramos que de fato não precisamos da aleatoridade para construirmos eficientes esquemas de *hashing*.

Esquema	BMZ	D-BMZ	J-BMZ	D-J-BMZ
Perfeito?	sim	sim	sim	sim
Mínimo?	sim	sim	sim	sim
Prático?	sim	sim	sim	sim
Determinístico?	não	sim	não	sim
Tempo do melhor caso	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Tempo do pior caso	$+\infty$	$O(n^{2L\|\Sigma\|+1})$	$+\infty$	$O(n^{2L\|\Sigma\|+1})$
Tempo do caso médio	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Tamanho da *hash*	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Esquema	BDZ	D-BDZ	J-BDZ	D-J-BDZ
Perfeito?	sim	sim	sim	sim
Mínimo?	sim	sim	sim	sim
Prático?	sim	sim	sim	sim
Determinístico?	não	sim	não	sim
Tempo do melhor caso	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Tempo do pior caso	$+\infty$	$O(n)$	$+\infty$	$O(n)$
Tempo do caso médio	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Tamanho da *hash*	$2{,}62n + O(1)$	$2{,}62n + O(1)$	$2{,}62n + 32$	$2{,}62n + 32$

QUADRO 4.1 — Comparação dos esquemas de *hashing* abordados.

A grande vantagem que nossas versões oferecem é o comportamento determinístico. Para uma mesma entrada, um de nossos esquemas fornecerá sempre a mesma saída. Acreditamos que isso possa ser útil para a construção de um esquema dinâmico de *hashing*, um esquema em que chaves podem ser inseridas no conjunto de dados ou excluídas dele. Tal como o esquema FKS, determinístico, inspirou um esquema dinâmico, motivamos trabalhos futuros a tentarem a construção de um esquema dinâmico baseado nos excelentes esquemas de Botelho, Kohayakawa e Ziviani

(2005) e de Botelho, Pagh e Ziviani (2007), na busca de estruturas de dados mais eficientes e práticas.

A despeito da relevância do trabalho de Botelho, Kohayakawa e Ziviani (2005), com a demonstração que nós oferecemos no Teorema 3.1 (p. 109) para a conjectura proposta pelos primeiros autores, encerramos a fundamentação teórica dum esquema que já era muito prestigiado pela comunidade. Esperamos que nosso trabalho possa ser útil para os pesquisadores nessa empreitada.

Estruturas de dados eficientes definitivamente não são um assunto esgotado.

REFERÊNCIAS

ABRAMOWITZ, M.; STEGUN, I. A. *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*. Nova Iorque, EUA: Dover Publications, 1972.

AHO, A. V.; LEE, D. Storing a dynamic sparse table. In: FOCS86. *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*. Toronto, Canadá: IEEE CSP, 1986. p. 55–60.

ALON, N. et al. Linear hash functions. *J. ACM*, p. 667–683, 1999.

ALON, N.; NAOR, M. Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, v. 16, p. 16–434, 1996.

BERGE, C. *Graphs and Hypergraphs*. Amsterdã: North Holland, 1973.

BOTELHO, F.; KOHAYAKAWA, Y.; ZIVIANI, N. A practical minimal perfect hashing method. In: WEA05. *Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms*. Ilha Santorini, Grécia: Springer, 2005. (LNCS, v. 3503), p. 488–500.

BOTELHO, F.; PAGH, R.; ZIVIANI, N. Simple and space-efficient minimal perfect hash functions. In: WADS07. *Proceedings of the 10th Workshop on Algorithms and Data Structures*. Halifax, Canadá: Springer, 2007. (LNCS, v. 4619), p. 139–150.

BRODNIK, A.; MUNRO, J. I. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, v. 28, n. 5, p. 1627–1640, 1999.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. *Introduction to Algorithms*. Cambridge, Inglaterra: MIT Press, 1990.

CZECH, Z. J.; HAVASB, G.; MAJEWSKI, B. S. Fundamental study perfect hashing. *Theoretical Computer Science*, p. 1–143, 1997.

DIESTEL, R. *Graph Theory*. Segunda edição. Nova Iorque, EUA: Springer-Verlag, 2000. ISBN 0-387-98976-5; 0-387-95014-1.

DIETZFELBINGER et al. Dynamic perfect hashing: Upper and lower bounds. In: FOCS88. *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*. Toronto, Canadá: IEEE PR, 1988. p. 524–531.

DIETZFELBINGER, M.; WEIDLING, C. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, p. 47–68, 2007.

ERDÖS, P.; RÉNYI, A. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, v. 5, n. 187, p. 17–60, 1960.

FOX, E. A.; CHEN, Q. F.; HEATH, L. S. A faster algorithm for constructing minimal perfect hash functions. In: ACM. *Proceedings of the 15th Annual International Conference on Research and Development in Information Retrieval, Data Structures*. Dublin, Irlanda: ACM, 1992. p. 266–273.

FREDMAN, M. L.; KOMLÒS, J. On the size of separating systems and families of perfect hash functions. *SIAM J. Alg. Disc. Meth.*, v. 5, n. 1, p. 61–68, 1984. ISSN 0196-5212.

FREDMAN, M. L.; KOMLÒS, J.; SZEMERÉDI, E. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, v. 31, n. 3, p. 538–544, 1984.

GRAHAM, R. L.; KNUTH, D. E.; PATASHNIK, O. *Concrete Mathematics*. Segunda edição. Boston, EUA: Addison-Wesley, 1994. ISBN 0-201-55802-5.

HAGERUP, T.; THOLEY, T. Efficient minimal perfect hashing in nearly minimal space. In: STACS01. *Proceedings of the 18th Symposium on Theoretical Aspects of Computer Science*. Dresden, Alemanha: Springer, 2001. (LNCS, v. 2010), p. 317–326.

HAVAS, G. et al. Graphs, hypergraphs and hashing. In: WG1993. *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*. Utrecht, Holanda: Springer, 1993. (LNCS, v. 790), p. 153–165. ISBN 3-540-57899-4.

JENKINS, B. Hash functions. *Dr. Dobbs Journal*, p. 107–109, set. 1997.

MELHORN, K. *Data Structures and Algorithms I: Sorting and Searching*. Berlim, Alemanha: Springer, 1984.

PAGH, R. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, v. 31, p. 2001, 2002.

PAGH, R.; RODLER, F. F. Cuckoo hashing. *Journal of Algorithms*, Elsevier, v. 51, p. 122–144, 2004.

PAPADIMITRIOU, C. *Computational complexity*. Boston, EUA: Addison-Wesley, 1995.

PITTEL, B.; WORMALD, N. C. Counting connected graphs inside-out. *J. Comb. Th. B*, v. 93, p. 127–172, 2005.

RADHAKRISHNAN, J. Improved bounds for covering complete uniform hypergraphs. *Information Processing Letters*, v. 41, n. 4, p. 203–207, mar. 1992. ISSN 0020-0190.

SCHMIDT, J. P.; SIEGEL, A. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, v. 19, n. 5, p. 775–786, 1990.

SILBERSCHATZ, A.; GALVIN, P. B. *Sistemas Operacionais: Conceitos e Aplicações*. Nova Jersey, EUA: Prentice Hall, 2000.

APÊNDICES

A LEMATA

Para não atrapalharem a fluidez da leitura do texto, optamos por transformar as partes mais técnicas de algumas demonstrações da presente dissertação em lemas, os quais reunimos todos neste apêndice, organizados pelos capítulos em que foram utilizados pela primeira vez.

A.1 CAPÍTULO 1 — INTRODUÇÃO

NOTAÇÃO A.1: Sendo A um conjunto qualquer e $n \leqslant |A|$ um natural, $\binom{A}{n}$ denota o conjunto de todos os subconjuntos de A de tamanho n.

$$\binom{A}{n} = \{B \subseteq A : |B| = n\}. \tag{A.2}$$

DEFINIÇÃO A.3 (Classe perfeita de *hashes*): Dados um universo arbitrário U de tamanho u, uma tabela *hash* M de tamanho m e um natural n, uma classe H de funções $h\colon U \to M$ é dita (U, M, n)-*perfeita* se, para todo conjunto $S \subseteq U$ com n chaves, houver alguma *hash* $h_S \in H$ perfeita para S.

LEMA A.4: *Dados um universo arbitrário U de tamanho u, uma tabela hash M de tamanho m e um natural n, e sendo H uma classe de hashes (U, M, n)-perfeita, então,*

$$|H| \geqslant \frac{\binom{u}{n}}{\left(\frac{u}{m}\right)^n \binom{m}{n}}. \tag{A.5}$$

Demonstração: Tomemos uma *hash* h sobre U e n posições distintas $i_1, i_2, \ldots, i_n$ da tabela *hash*. Tomemos

$$\mathscr{S}^h_{i_1,\ldots,i_n} = \big\{S \subseteq U, |S| = n \colon h(S) = \{i_1, \ldots, i_n\}\big\}. \tag{A.6}$$

Note-se que h é perfeita para todo $S \in \mathscr{S}^h_{i_1,\ldots,i_n}$. Assim, como escolher um $(x_1, \ldots, x_n)$ em $h^{-1}(1) \times \cdots \times h^{-1}(n)$ significa escolher um $S = \{x_1, \ldots, x_n\}$ em $\mathscr{S}^h_{i_1,\ldots,i_n}$, temos que

$$\big|\mathscr{S}^h_{i_1,\ldots,i_n}\big| \leqslant |h^{-1}(1)| \cdots |h^{-1}(n)|. \tag{A.7}$$

Agora, seja $\mathscr{S}^h$ o conjunto de todos os conjuntos com n chaves para os quais h seja perfeita. Como

$$\mathscr{S}^h = \bigcup_{\{i_1,\ldots,i_n\} \in \binom{M}{n}} \mathscr{S}^h_{i_1,\ldots,i_n}, \tag{A.8}$$

temos que

$$\big|\mathscr{S}^h\big| \leqslant \sum_{\{i_1,\ldots,i_n\} \in \binom{M}{n}} |h^{-1}(1)| \cdots |h^{-1}(n)|. \tag{A.9}$$

Como $\{h^{-1}(1), \ldots, h^{-1}(n)\}$ é uma partição de U, $|h^{-1}(1)| \cdot \cdots \cdot |h^{-1}(n)| \leqslant \left(\frac{u}{m}\right)^n$, e, consequentemente,

$$\big|\mathscr{S}^h\big| \leqslant \left(\frac{u}{m}\right)^n \binom{m}{n}. \tag{A.10}$$

A quantidade de *hashes* que são perfeitas para algum conjunto S é no mínimo a quantidade de conjuntos S dividida pelo máximo de conjuntos para os quais uma mesma *hash* h é perfeita. Dessarte,

$$|H| \geqslant \frac{\big|\big\{S \subseteq U \colon |S| = n\big\}\big|}{\max_{h \in H}\big|\mathscr{S}^h\big|} \geqslant \frac{\binom{u}{n}}{\left(\frac{u}{m}\right)^n \binom{m}{n}}, \tag{A.11}$$

como queríamos mostrar. □

LEMA A.12: *Dados um universo arbitrário U de tamanho u, uma tabela hash M de tamanho m e um natural n, e sendo H uma classe de hashes (U, M, n)-perfeita, então,*

$$|H| \geqslant \frac{\log u}{\log m}. \tag{A.13}$$

Demonstração: Seja $t = |H|$ e $H = \{h_1, \ldots, h_t\}$. Sejam $U_0, \ldots, U_t$ os $t+1$ subconjuntos de U definidos por:

$$U_i = \begin{cases} U, & \text{se } i = 0; \\ U_{i-1} \cap h_i^{-1}(j_i), & \text{se } 0 < i \leqslant t, \end{cases} \tag{A.14}$$

sendo j_i um elemento da tabela *hash* de maior imagem inversa por h_i. Então, $h_i(x) = j_i$ para todo $x \in U_i$. Assim, para todo conjunto S com pelo menos 2 chaves em U_1, a *hash* h_1 não é perfeita para S. Indutivamente, para $i \in \{1, \ldots, t\}$ e todo conjunto S com pelo menos 2 chaves em U_i, as *hashes* $h_1, \ldots, h_i$ não são perfeitas para S. Logo, precisamos ter $|U_t| \leqslant 1$; do contrário, qualquer S que tivesse ao menos 2 chaves em U_t não encontraria *hash* perfeita em H, e H não seria (U, M, n)-perfeita, por definição. Ora,

$$|U_i| \geqslant \frac{|U_{i-1}|}{m} \geqslant \frac{|U_{i-2}|}{m^2} \geqslant \cdots \geqslant \frac{|U_0|}{m^i}. \tag{A.15}$$

Dessarte,

$$|U_t| \geqslant 1 \geqslant \frac{u}{m^t}, \tag{A.16}$$

o que implica que $t \geqslant \frac{\log u}{\log m}$, como queríamos mostrar. □

LEMA A.17: *Dados um universo arbitrário U de tamanho u, uma tabela hash M de tamanho m e um natural n, existe ao menos um conjunto $S \subseteq U$ de n chaves tal que a hash perfeita para S de menor representação tem de tamanho no mínimo*

$$\max\left(\frac{n(n-1)}{2m\ln 2} - \frac{n(n-1)}{2u\ln 2}\left(1 - \frac{n-1}{u}\right), \log\log u - \log\log m\right) - 1 \tag{A.18}$$

bits.

Demonstração: Sejam

$$b_1 = \frac{\binom{u}{n}}{\left(\frac{u}{m}\right)^n \binom{m}{n}}, \qquad \text{e} \qquad b_2 = \frac{\log u}{\log m}. \tag{A.19}$$

Sendo H uma classe de *hashes* (U, M, n)-*perfeita*, sabemos, dos Lemas A.4 e A.12, que $|H| \geqslant \max(b_1, b_2)$. Como há no máximo 2^w cadeias binárias com tamanho no máximo w, existe ao menos um $S \subseteq U$ de n chaves tal que a *hash* perfeita para S de menor representação tem tamanho no mínimo $\max(\lg b_1, \lg b_2) - 1 \geqslant \max(\log b_1, \log b_2) - 1$, como queríamos mostrar. □

A.2 CAPÍTULO 2 — RESENHA LITERÁRIA

LEMA A.20: *Sejam $U = \{0, \ldots, u-1\}$ e $S \subseteq U$, $|S| = n$. Se $m \geqslant n$, então, existe um $a \in U \setminus \{0\}$ para o qual a hash $h\colon U \to \{0, \ldots, m-1\}$ definida pela Equação 2.3 (p. 39) é tal que*

$$\sum_{i=0}^{m-1} \binom{|S_i|}{2} < \frac{n(n-1)}{m}. \tag{A.21}$$

Demonstração: Dado um a qualquer em $\{1, \ldots, u-1\}$, definimos

$$h_a\colon x \to (ax \bmod u) \bmod m \tag{A.22}$$

e $S_i^{(a)}$ o *bucket* de colisão (Definição 1.10, p. 27) para o endereço i da tabela da *hash* h_a. Sabemos que $\binom{S_i^{(a)}}{2} = \{\{x, y\} : x, y \in S, x \neq y, h_a(x) = h_a(y) = i\}$. Se

$x \neq y$ mas $h_a(x) = h_a(y) = i$, então, $(ax \bmod u) \bmod m = (ay \bmod u) \bmod m$. Portanto, $a(x-y) \bmod u \in X = \{m, 2m, 3m, \ldots, u-m, u-2m, u-3m, \ldots\}$, e

$$\begin{aligned} \sum_{a=1}^{u-1} \sum_{i=0}^{m-1} \binom{|S_i^{(a)}|}{2} &= \sum_{\substack{x,y \in S \\ x \neq y}} |\{a : a(x-y) \bmod u \in X\}| \\ &= |\{a : \exists (x,y), x \neq y, a(x-y) \bmod u \in X\}|. \end{aligned} \tag{A.23}$$

Uma vez que, para todo par (x, y) tal que $x \neq y$ vale que $|\{a : h_a(x) = h_a(y)\}| < \frac{2u}{m}$,

$$\sum_{a=1}^{u-1} \sum_{i=0}^{m-1} \binom{|S_i^{(a)}|}{2} < \binom{n}{2} \frac{2u}{m} = \frac{un(n-1)}{m}. \tag{A.24}$$

Logo, existe ao menos um a tal que $\sum_i \binom{|S_i^{(a)}|}{2} < \frac{n(n-1)}{m}$. □

LEMA A.25: $\lim_{x \to 0} \ln(1-x) = -x$.

Demonstração: É muito conhecida a série

$$\ln(1+x) = \sum_{n \geqslant 1} (-1)^{n+1} \frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots, \tag{A.26}$$

apresentada, por exemplo, no trabalho de Abramowitz e Stegun (1972). Ora, temos então que

$$\ln(1-x) = (-x) - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \cdots = (-x) - \left(\sum_{n \geqslant 2} \frac{x^n}{n} \right). \tag{A.27}$$

Porém,

$$\lim_{x \to 0} \sum_{n \geqslant 2} \frac{x^n}{n} = 0, \tag{A.28}$$

e chegamos ao resultado esperado. □

LEMA A.29: *No Algoritmo 2.58 (p. 67) para um grafo G_{crit}, todo vértice $v \in V(G_{\text{crit}})$ é enfileirado uma única vez, por ser o vértice inicial da busca ou por causa de um vizinho seu v_0, e desenfileirado também uma única vez.*

Demonstração: Das linhas 2 e 15–17, percebemos que só são enfileirados vértices que ainda não tenham sido rotulados. Ora, como um vértice, ao ser desenfileirado, sempre é rotulado (linhas 4–14), a única possibilidade restante de um vértice ser enfileirado uma segunda vez é ele ainda não ter sido desenfileirado. Mas isso nunca acontece, pois, de acordo com as linhas 2 e 15, se um vértice é enfileirado, seguramente não está na fila.

Como G_{crit} é conexo, e como u_0 seguramente é enfileirado, também é impossível também é impossível que haja algum vértice v que não tenha sido enfileirado vez alguma, pois só não teria sido enfileirado por algum vizinho seu v_0 por ocasião de v já ter sido rotulado. Mas não há como um vértice ser rotulado sem entrar na fila, uma vez que nenhum vértice de G_{crit} está rotulado no início. □

LEMA A.30: *Após a execução do Algoritmo 2.58 (p. 67) para um grafo G_{crit}, é uma árvore o subgrafo $T = (V(G_{\text{crit}}), E(T))$ de G_{crit} definido por:*

$$\{u_1, u_2\} \in T \qquad \textit{se, e somente se, } u_1 \textit{ enfileirou } u_2 \textit{ ou } u_2 \textit{ enfileirou } u_1. \qquad \textit{(A.31)}$$

Demonstração: Tomemos um vértice u_2 que não seja o vértice inicial da busca. Podemos assumir que ele existe já que, se $V(G_{\text{crit}})$ contivesse apenas o vértice inicial da busca, $\delta(G_{\text{crit}})$ não seria no mínimo 2. Do Lema A.29, existe apenas um u_1 vizinho de u_2 em T que enfileirou u_2, o que implica em todos os outros vizinhos de u_2 em T serem enfileirados por u_2.

Se u_2 não possui mais vizinhos, seguramente u_2 não pertence a um ciclo. Con-

sideremos, então, o caso em que u_2 possui outros vizinhos que não u_1 e suponhamos que u_2 pertence a um ciclo.

a) Se u_1 também pertence a esse ciclo, há caminho $v_0, \ldots, v_\ell = u_1$ em T que não passa por u_2 entre u_1 e algum vizinho $v_0 \neq u_1$ de u_2. Lembrando que todo vértice só possui um vizinho em T que o enfileirou, e já que foi u_2 quem enfileirou v_0, temos, indutivamente, que todo $v_j, j \neq 0$, foi enfileirado por v_{j-1}, e, portanto, que u_1 foi enfileirado por $v_{\ell-1}$, o que é um absurdo, dado que $v_{\ell-1}$ entrou na fila depois de u_2 e dado que u_1 entrou antes de u_2.
b) Se, por outro lado, u_1 não pertence a esse ciclo, há um caminho $v_0, \ldots, v_\ell$ em T que não passa por u_2 entre dois vizinhos v_0 e v_ℓ distintos de u_1 e distintos entre si. Assim sendo, como foi u_2 quem enfileirou v_0, foi, indutivamente, $v_{\ell-1} \neq u_2$ quem enfileirou v_ℓ, um absurdo, pois u_2 que enfileirou v_ℓ.

Desse modo concluímos que T não possui ciclos. □

DEFINIÇÃO A.32: Chamamos de *aresta de retorno* toda aresta de $E(G_{\text{crit}})$ que não pertence a $E(T)$, sendo T como no enunciado do Lema A.30 (p. 140). Ademais, usamos N_{bedges} para denotar o número de arestas de retorno em $E(G_{\text{crit}})$.

LEMA A.33: $N_{\text{bedges}} = |E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1$.

Demonstração: O número de arestas de toda árvore é o número de vértices da árvore mais 1. Assim, como $N_{\text{bedges}} + |E(T)| = |E(G_{\text{crit}})|$, e como $V(T) = V(G_{\text{crit}})$,

$$N_{\text{bedges}} + |V(G_{\text{crit}})| + 1 = |E(G_{\text{crit}})|, \tag{A.34}$$

como queríamos mostrar. □

LEMA A.35: $\max A_E \leqslant 2|V(G_{\text{crit}})| - 3 + 2N_{\text{t}}$.

Demonstração: N_{t} é o número de reassociações feitas pelo Algoritmo 2.58 (p. 67) no processo de rotulação dos vértices de G_{crit}. Cada vértice recebe um rótulo i, que inicialmente vale 0 e é incrementado ou a cada rotulação ou a cada reassociação. Como exatas $|V(G_{\text{crit}})|$ rotulações são feitas, i é incrementado $|V(G_{\text{crit}})| + N_{\text{t}}$ vezes. Assim, o maior rótulo atribuído a algum vértice crítico é $|V(G_{\text{crit}})| + N_{\text{t}} - 1$, e o segundo maior rótulo, consequentemente, é no máximo $|V(G_{\text{crit}})| + N_{\text{t}} - 2$. A_E é o conjunto de todas as somas $g(u) + g(v)$, para toda aresta crítica $\{u, v\}$. Finalmente, $\max A_E \leqslant |V(G_{\text{crit}})| + N_{\text{t}} - 1 + |V(G_{\text{crit}})| + N_{\text{t}} - 2$, o que nos conduz ao resultado esperado. □

A.3 CAPÍTULO 3 — METODOLOGIA E RESULTADOS

LEMA A.36: *Sempre que tentamos rotular com algum j alguma aresta da árvore de busca do Algoritmo 2.58 (p. 67), j é maior que qualquer rótulo de qualquer outra aresta da árvore.*

Demonstração: Se estamos tentando rotular com j uma aresta e da árvore de busca, é porque estamos tentando rotular algum vértice $v \in e$ com $i = j - g(w)$, sendo w o outro vértice de e, que, evidentemente, já precisa estar rotulado. Por contradição, suponhamos que existe alguma aresta da árvore já rotulada $\{u_1, u_2\}$ com $g(u_1) < g(u_2)$ tal que $g(u_1) + g(u_2) \geqslant g(w) + i$. Como i é maior que qualquer rótulo de qualquer vértice rotulado, temos que

$$g(w) < g(u_1) < g(u_2) < i. \tag{A.37}$$

Porque um vértice só é rotulado ao ser desenfileirado, w foi desenfileirado antes de u_1,

que foi desenfileirado antes de u_2, que foi desenfileirado antes de v. Já que estamos tratando duma fila, w foi enfileirado antes de u_1, que foi enfileirado antes de u_2, que foi enfileirado antes de v.

De acordo com o resultado do Lema A.29, foi w *o* vértice que enfileirou v, assim como foi u_1 *o* vértice que enfileirou u_2. Dessarte, quando u_2 foi enfileirado, u_1 já tinha sido desenfileirado, e, portanto, w também já tinha sido desenfileirado, e, ainda, v já tinha sido enfileirado. Mas v não pode ter sido enfileirado antes de u_2. Logo, a suposta aresta $\{u_1, u_2\}$ não existe na árvore. □

LEMA A.38: *Sendo x um real em $(0,1)$ e n um natural qualquer,* $\lim_{n\to\infty} \sum_{i=1}^{n} i x^{i-1} = \frac{1}{(1-x)^2}$.

Demonstração: É muito conhecido que, para todo real x no intervalo $(0,1)$,

$$\lim_{n\to\infty} \sum_{i=0}^{n} x^i = \frac{1}{1-x}. \tag{A.39}$$

Portanto,

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\lim_{n\to\infty} \sum_{i=0}^{n} x^i\right) = \frac{\mathrm{d}}{\mathrm{d}x}\Big(\frac{1}{1-x}\Big), \tag{A.40}$$

e, consequentemente,

$$\lim_{n\to\infty} \sum_{i=1}^{n} i x^{i-1} = \frac{1}{(1-x)^2}, \tag{A.41}$$

já que

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\lim_{n\to\infty} \sum_{i=0}^{n} x^i\right) = \lim_{n\to\infty} \frac{\mathrm{d}}{\mathrm{d}x}\left(\sum_{i=0}^{n} x^i\right) = \lim_{n\to\infty} \sum_{i=1}^{n} i x^{i-1} \tag{A.42}$$

e

$$\frac{\mathrm{d}}{\mathrm{d}x}\Big(\frac{1}{1-x}\Big) = \frac{\mathrm{d}\big(\frac{1}{1-x}\big)}{\mathrm{d}(1-x)} \cdot \frac{\mathrm{d}(1-x)}{\mathrm{d}x} = \Big(-\frac{1}{(1-x)^2}\Big)(-1) = \frac{1}{(1-x)^2}, \tag{A.43}$$

como queríamos mostrar. □

LEMA A.44: *Sendo k um inteiro positivo qualquer,* $\lim_{n\to\infty} \sum_{i=1}^{n} i\big(1-\frac{1}{k}\big)^{i-1} = k^2$*.*

Demonstração: Se tomamos no Lema A.38 $x = 1 - \frac{1}{k}$, temos que

$$\lim_{n\to\infty} \sum_{i=1}^{n} i\Big(1-\frac{1}{k}\Big)^{i-1} = \frac{1}{\big(1-\big(1-\frac{1}{k}\big)\big)^2} = k^2 \tag{A.45}$$

e chegamos ao resultado esperado. □

B AS *HASHES* DE JENKINS

É muito comum querermos mapear uma cadeia de *bytes* numa palavra de 32 *bits* (4 *bytes*). Cadeias de *bytes* podem ser, por exemplo, *strings* de caracteres, já que, via de regra, caracteres são implementados como variáveis de um único *byte*, segundo uma tabela de caracteres, como a tabela ASCII. Palavras de 32 *bits* podem ser consideradas números naturais entre 0 e $2^{32} - 1$. Podemos considerar como um exemplo o caso em que queremos indexar um conjunto com n *strings* por uma variável inteira sem sinal num processador de 32 *bits*.

Jenkins (1997) propôs uma *hash* J_1 que mapeia uma cadeia K de no máximo $2^{32} - 1$ *bytes* numa palavra j_1 de 32 *bits* de modo que cada *bit* em K influencia cada *bit* em j_1; alternativamente, cada *bit* de j_1 é função de todos os *bits* de K. Dessarte, para casos reais de aplicação desse tipo de *hashing*, nos quais os conjuntos de cadeias — as chaves — frequentemente não seguem a distribuição uniforme, J_1 distribui bem as chaves na tabela *hash*, o conjunto $\{0, \ldots, 2^{32} - 1\}$, considerando as palavras de 32 *bits* como números naturais. Nos casos reais, frequentemente pode ocorrer de chaves se parecerem muito umas com as outras. No entanto, na *hash* de Jenkins, uma pequena mudança num *byte* da cadeia de entrada pode acarretar uma palavra de saída completamente diferente, o que auxilia na boa distribuição das chaves na tabela e reduz as colisões. Notemos, no entanto, que a *hash* de Jenkins não é perfeita nem mínima.

A *hash* de Jenkins também é muito eficientemente computável. Para uma cadeia K de tamanho $|A| = \ell$ *bytes*, o endereço j_1 associado à cadeia K pela *hash* J_1 é computado em tempo aproximadamente $6\ell + 35$, que pode ser considerado constante, já que limitamos ℓ superiormente por $2^{32} - 1$. Na prática, Jenkins observou que o tamanho das cadeias varia de 8 a 200 *bytes*.

ALGORITMO B.1 — Computação da *hash* de Jenkins.

ENTRADA: uma cadeia K de ℓ *bytes* e uma palavra s de 4 *bytes*.
SAÍDA: $J_1(K, s)$.
1: Complete K com *bits* nulos até que 12 divida ℓ.
2: $a \leftarrow \texttt{10011110001101110111100110111001}$.
3: $b \leftarrow \texttt{10011110001101110111100110111001}$.
4: $c \leftarrow s$.
5: ENQUANTO $\ell \neq 0$, FAÇA:
6: $\quad a \leftarrow a\texttt{+(}K\texttt{[1],}K\texttt{[2],}K\texttt{[3],}K\texttt{[4])}$;
7: $\quad b \leftarrow b\texttt{+(}K\texttt{[5],}K\texttt{[6],}K\texttt{[7],}K\texttt{[8])}$;
8: $\quad c \leftarrow c\texttt{+(}K\texttt{[9],}K\texttt{[10],}K\texttt{[11],}K\texttt{[12])}$;
9: $\quad \texttt{mix}(a, b, c)$ (Algoritmo B.5, p. 147);
10: $\quad K \leftarrow K \gg 12$;
11: $\quad \ell \leftarrow \ell - 12$,
12: FIM.
13: Devolva c.

A *hash* de Jenkins não é função unicamente da cadeia de *bytes*, mas também duma outra palavra s de 32 *bits*, que pode ser considerada como uma semente aleatória para o J_1. Descrevemos o procedimento para calcular $J_1(K, s)$ no Algoritmo B.1, sob a convenção das Notações B.2, B.3 e B.4.

NOTAÇÃO B.2: Sendo K uma cadeia de ℓ *bytes*, usamos $K\texttt{[}i\texttt{]}$ para denotar o i-ésimo *byte* da cadeia K, a qual pode ser entendida como uma concatenação das subcadeias $K\texttt{[1]}, \ldots, K\texttt{[}\ell\texttt{]}$. Ademais, sendo d um natural qualquer, usamos $K \gg d$ para denotar a cadeia de $8\ell - d$ *bits* obtida com o deslocamento dos *bits* de K d *bits* para a direita, descartando-se os últimos d *bits* de K.

NOTAÇÃO B.3: Sendo w_1 e w_2 duas palavras de 32 *bits*, a soma w_1+w_2 é a palavra formada pelos primeiros 32 *bits* da representação binária de $n_1 + n_2$, sendo n_1 e n_2 os naturais representados por w_1 e por w_2, respectivamente. Analogamente, definimos w_1-w_2 como a representação binária de $|n_1 - n_2|$. Ainda definimos $w_1\hat{}w_2$ como a avaliação *bit* a *bit* do operador lógico de disjunção exclusiva — *xor* — para as palavras w_1 e w_2.

NOTAÇÃO B.4: Sendo w uma palavra de 4 *bytes*, e sendo b_1, b_2, b_3 e b_4 outros quatro *bytes*, usamos w+(b_1,b_2,b_3,b_4) para denotar w+b, sendo b a palavra de 4 *bytes* obtida através da concatenação de b_1, b_2, b_3 e b_4.

Como se pode perceber, a execução do Algoritmo B.1 é uma interação entre três palavras de 4 *bytes* a, b e c. Inicialmente, a e b recebem um valor qualquer, escolhido pelo autor como a codificação em ponto flutuante de 32 *bits* da razão áurea ($\varphi = \frac{1+\sqrt{5}}{2}$), na qual os *bits* iguais a 0 e os *bits* iguais a 1 se distribuem sem um padrão definível, valendo como uma boa sequência semialeatória de *bits*. Em seguida, lemos os *bytes* de K de 12 em 12. A cada 12 *bytes*, conforme a Notação B.4, os 4 primeiros *bytes* são somados com a, os 4 *bytes* seguintes são somados com b, e os outros 4 *bytes* são somados com c. Como todo *bit* de K precisa interferir em todo *bit* de c, que será devolvido como $J_1(K,r)$, as palavras a, b e c passam por um procedimento chamado `mix`, descrito pelo Algoritmo B.5, satisfazendo a Propriedade B.6.

ALGORITMO B.5 — `mix`(a,b,c).

1: $a \leftarrow a$-b, $a \leftarrow a$-c e $a \leftarrow a$^$c \gg 13$.
2: $b \leftarrow b$-c, $b \leftarrow b$-a e $b \leftarrow b$^$c \ll 8$.
3: $c \leftarrow c$-a, $c \leftarrow c$-b e $c \leftarrow c$^$c \gg 13$.
4: $a \leftarrow a$-b, $a \leftarrow a$-c e $a \leftarrow a$^$c \gg 12$.
5: $b \leftarrow b$-c, $b \leftarrow b$-a e $b \leftarrow b$^$c \ll 16$.
6: $c \leftarrow c$-a, $c \leftarrow c$-b e $c \leftarrow c$^$c \gg 5$.
7: $a \leftarrow a$-b, $a \leftarrow a$-c e $a \leftarrow a$^$c \gg 3$.
8: $b \leftarrow b$-c, $b \leftarrow b$-a e $b \leftarrow b$^$c \ll 10$.
9: $c \leftarrow c$-a, $c \leftarrow c$-b e $c \leftarrow c$^$c \gg 15$.

PROPRIEDADE B.6: *Se inicialmente os bits de a, b e c são quase todos nulos ou uniformemente distribuídos:*

a) após a execução de `mix`(a,b,c)*, ao menos 32 bits entre todos os bits de a, b e c têm probabilidade no mínimo $\frac{1}{4}$ de haverem mudado;*

b) considerando-se uma sequência de execuções de `mix`(a,b,c)*, cada bit de c*

muda ao menos uma vez entre $\frac{1}{3}$ a $\frac{2}{3}$ do tempo.

Note-se que no Algoritmo B.5 abusamos da Notação B.2, pois, a rigor, $a \gg d$ seria uma palavra de $32 - d$ *bits*. No entanto, simplesmente completamos com 0 as d primeiros *bits* de $a \gg d$ para que tenhamos ainda 32 *bits*. Analogamente, também usamos $a \ll d$ para denotar a palavra de 32 *bits* obtida com o deslocamento de a d *bits* para a esquerda, descartando-se os d primeiros *bits* de a e completando-se com 0 os novos *bits* que surgem à direita.

As palavras a e b, computadas junto com a palavra c, bem que poderiam também ser fornecidos como endereços para K. Na realidade, o Algoritmo B.1 não computa apenas uma *hash* J_1, mas três *hashes*, de modo que $J_1(K, r) = c$, $J_2(K, r) = b$, e $J_3(K, r) = a$. As *hashes* J_2 e J_3 desfrutam das mesmas propriedades que a *hash* J_1.

Jenkins (1997) expõe uma implementação eficiente de seu algoritmo em linguagem C de modo que a computação de J_1, J_2, J_3 para uma cadeia K com ℓ *bits* requer a execução de apenas $6\ell + 35$ instruções. Além da escolha da razão áurea para os valores iniciais de a e b, com o objetivo de que se mapeiem todos os *bits* nulos para todos os *bits* nulos, Jenkins (1997) ainda usa vários outros truques de manipulação de registradores que permitem computação em paralelo de algumas instruções. As *hashes* de Jenkins, propostas para um problema bem específico mas muito comum, combinam grande eficiência em tempo e espaço com boa distribuição das chaves na tabela *hash*, considerando especialmente casos reais de conjuntos de chaves e os problemas que as muitas similitudes entre chaves poderiam causar. Por isso, são muito utilizadas em contextos para os quais importa mais a velocidade de computação e a praticidade da *hash* que a garantia absoluta da ausência de colisões.

C NOTAÇÃO ASSINTÓTICA E ANÁLISE DE ALGORITMOS

Um *algoritmo* é uma instância de um modelo computacional para computar a solução de um problema computacional. Assim, consideramos como um algoritmo um programa RAM (*Random Access Machine*), uma máquina de Turing, uma família de circuitos booleanos etc. Quando analisamos a complexidade de tempo ou de espaço de um algoritmo, é de suma importância que deixemos claro o modelo computacional adotado, já que diferentes modelos computacionais podem implicar diferentes complexidades. No presente trabalho, assumimos que os algoritmos são abstrações de alto nível do modelo RAM.

C.1 Análise de tempo e de espaço de algoritmos

No modelo RAM, o *tempo* de um algoritmo A para uma entrada X, denotado por $T_A(X)$, é o número de instruções que A executa na computação da saída correspondente para X. Se conseguimos mostrar que, para toda entrada possível X, $T_A(X)$ é uma função t do tamanho da representação binária da entrada (denotado por $|X|$), dizemos que $t(|X|)$ é a *complexidade de tempo* do algoritmo A. Estendemos essa definição e, por abuso, dizemos que $\mathscr{C}(n)$ é a complexidade de tempo de A, sendo $\mathscr{C}(n)$ uma classe de funções de um natural n, se, para toda entrada X de tamanho n, $T_A(X) = f(n)$ para algum $f \in \mathscr{C}$.

Ainda no modelo RAM, o *espaço* de um algoritmo A para uma entrada X, denotado por $S_A(X)$, é o número de *bits* que A utiliza na computação da saída correspondente para X, incluindo o tamanho da representação da própria entrada X. Se

conseguimos mostrar que, para toda entrada possível X, $S_A(X)$ é uma função $s(|X|)$ do tamanho da entrada, dizemos que s é a *complexidade de espaço* do algoritmo A. Estendemos essa definição e, por abuso, dizemos que $\mathscr{C}(n)$ é a complexidade de espaço de A se, para toda entrada X de tamanho n, $S_A(X) = f(n)$ para algum $f \in \mathscr{C}$.

A Análise Assintótica, disciplina da Matemática que estuda o crescimento de funções, agrupa funções em classes de acordo com seu comportamento assintótico. Essas classes são muito úteis para a Análise de Algoritmos não apenas por simplificarem algumas expressões matemáticas, mas especialmente porque, como classes de funções, podem servir de complexidade computacional tanto de tempo quanto de espaço.

C.2 Notação assintótica

Na Figura C.1 (p. 151), temos um esboço dos gráficos de duas funções de n: $100n$ e n^2. Embora n seja natural, esboçamos os gráficos das funções como se seus domínios fossem contínuos, para que fique mais claro o que queremos ilustrar. Note-se que, para $n < 100$, a função $100n$ é maior que a função n^2. No entanto, a partir de $n = 100$, n^2 se torna cada vez maior que a função $100n$. Na realidade, $\lim_{n\to\infty} \frac{100n}{n^2} = 0$. Ainda que, para alguns elementos do domínio, $100n$ seja maior que n^2, notamos que a função n^2 domina a função $100n$.

DEFINIÇÃO C.2: Dizemos que uma função $f\colon \mathbb{R} \to \mathbb{R}$ *domina* uma função $g\colon \mathbb{R} \to \mathbb{R}$, e escrevemos $f \gg g$, se $\lim_{x\to\infty} \frac{g(x)}{f(x)} = 0$. Do mesmo modo, dizemos que g *é dominada por* f e escrevemos $g \ll f$.

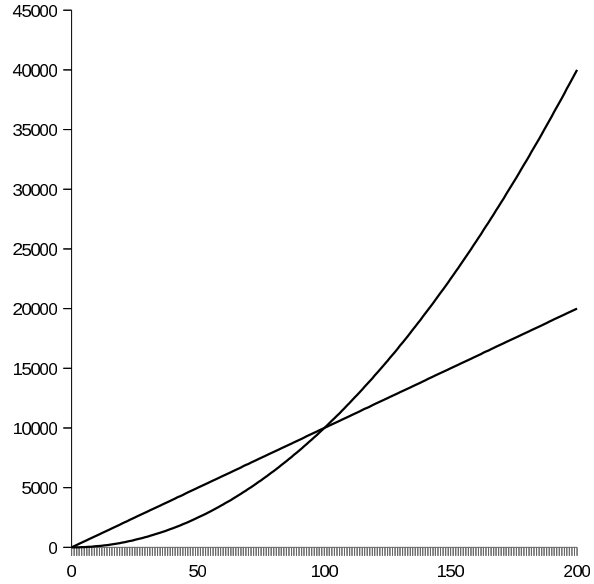


FIGURA C.1 — $n^2 \gg n$.

Assim, com as notações $\gg$ e $\ll$, estabelecemos uma hierarquia entre funções de acordo com o seu crescimento assintótico. Na Figura C.4 (p. 152), esboçamos os gráficos de algumas funções para ilustrarmos que $\ln n \ll n \ll n^2 \ll 2^n$.

Dada uma função $g(n)$, podemos definir uma classe de todas as funções f que são dominadas por g. Essa classe denotamos por $o(g(n))$, como na Definição C.3. Embora a Definição C.3 e as demais definições a seguir possam ser estendidas para funções com domínio nos reais, limitar-nos-emos às funções com domínio nos naturais, já que, em Análise de Algoritmos, estamos tratando de tamanhos de entradas.

DEFINIÇÃO C.3: Sendo duas funções $f, g\colon \mathbb{N} \to \mathbb{R}$, escrevemos $f(n) = o(g(n))$ quando $f(n) \ll g(n)$. Analogamente, escrevemos $f(n) = \omega(g(n))$ quando $f(n) \gg g(n)$.

Fica evidente que, na Definição C.3, o uso do símbolo $=$ é um abuso de notação. Seria muito mais apropriado o uso do símbolo $\in$, já que $o(g(n))$ denota, na realidade, uma classe de funções. Entretanto, o uso do símbolo $=$ já está por de-

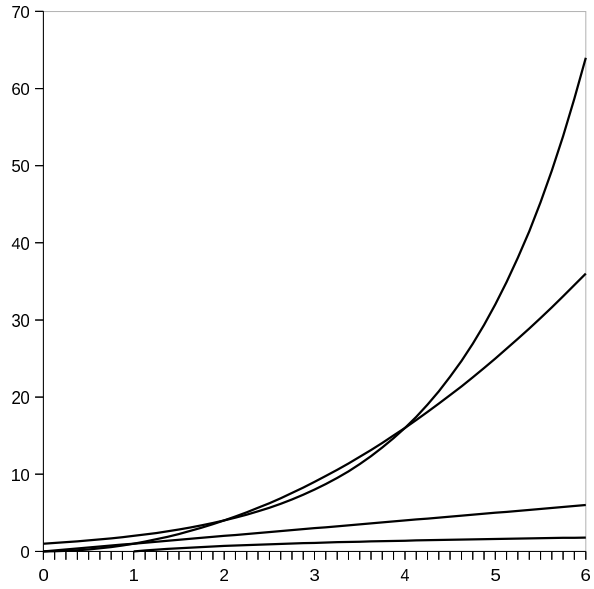


FIGURA C.4 — $\ln n \ll n \ll n^2 \ll 2^n$.

mais consagrado pela comunidade científica, por permitir que escrevamos expressões como:

$$s(n) = 50n^2 + o(n^2) + o(n). \tag{C.5}$$

De fato, a Equação C.5 é muito mais conveniente do que a Equação C.6:

$$s(n) = 50n^2 + f(n) + g(n) \qquad \text{para algum } f \in o(n^2) \text{ e algum } g \in o(n). \tag{C.6}$$

Para evitarmos confusão, convém-se que se leia o símbolo $=$ como "*é*", não como "*é igual a*". Assim, lemos $f(n) = o(g(n))$ como "$f(n)$ *é* $o(g(n))$", nunca como "$f(n)$ *é igual a* $o(g(n))$".

Alternativamente, podemos definir $o(g(n))$ e $\omega(g(n))$ como se segue.

DEFINIÇÃO C.7: Sendo duas funções $f, g\colon \mathbb{N} \to \mathbb{R}$, escrevemos $f(n) = o(g(n))$ quando,

para todo real não-nulo c, existe um natural n_0 tal que

$$|f(n)| \leqslant c|g(n)| \qquad \text{para todo } n \geqslant n_o. \tag{C.8}$$

Analogamente, escrevemos $f(n) = \omega(g(n))$ quando, para todo real não-nulo c, existe um natural n_0 tal que

$$|f(n)| \geqslant c|g(n)| \qquad \text{para todo } n \geqslant n_o. \tag{C.9}$$

Note-se que $f(n) = \omega(g(n))$ quando $g(n) = o(f(n))$ e vice-versa.

A equivalência entre as Definições C.3 e C.7 segue naturalmente da definição formal de limite. Na Definição C.10, outras definições assintóticas importantes apresentamos.

DEFINIÇÃO C.10: Sendo duas funções $f, g\colon \mathbb{N} \to \mathbb{R}$, escrevemos $f(n) = O(g(n))$ quando existe um real não-nulo c e um natural n_0 tais que

$$|f(n)| \leqslant c|g(n)| \qquad \text{para todo } n \geqslant n_o. \tag{C.11}$$

Analogamente, escrevemos $f(n) = \Omega(g(n))$ quando existe um real não-nulo c e um natural n_0 tais que

$$|f(n)| \geqslant c|g(n)| \qquad \text{para todo } n \geqslant n_o. \tag{C.12}$$

Note-se que $f(n) = \Omega(g(n))$ quando $g(n) = O(f(n))$ e vice-versa.

Quando $f(n) = O(g(n))$, costumamos dizer que $f(n)$ é assintoticamente no máximo $g(n)$, ou que o crescimento assintótico de $f(n)$ é limitado superiormente por

$g(n)$, já que, a partir de algum n_0 no domínio, f nunca é maior que cg em valores absolutos, para alguma constante real c. Analogamente, quando $f(n) = \Omega(g(n))$, dizemos que $f(n)$ é assintoticamente no mínimo $g(n)$, ou que o crescimento assintótico de $f(n)$ é limitado inferiormente por $g(n)$.

Por exemplo, ambas as funções $n^2 + 23n$ e $n\sqrt{n+42}$ são $O(n^2)$, pois

$$|n^2 + 23n| \leqslant |n^2 + n \cdot n| \leqslant 2|n^2| \qquad \text{para todo } n \geqslant 23, \tag{C.13}$$

e

$$|n\sqrt{n+42}| \leqslant |n\sqrt{2n}| \leqslant (\sqrt{2})|n^2| \qquad \text{para todo } n \geqslant 42. \tag{C.14}$$

Curiosamente, $n^2 + 23n$ também é $\Omega(n^2)$, o que não ocorre com $n\sqrt{n+42n}$ pois, embora

$$|n^2 + 23n| \geqslant |n^2| \qquad \text{para todo } n \geqslant 0, \tag{C.15}$$

se assumimos que existem c e n_0 tais que, para todo $n \geqslant n_0$, $|n\sqrt{n+42}| \geqslant c|n^2|$, temos que, para todo $n \geqslant \max(n_0, 42)$, $cn \leqslant \sqrt{2n}$, e, portanto, que $\sqrt{n} \leqslant \frac{\sqrt{2}}{c}$, um absurdo.

DEFINIÇÃO C.16: Sendo duas funções $f, g\colon \mathbb{N} \to \mathbb{R}$, escrevemos $f(n) = \Theta(g(n))$ quando $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Dizemos quando $f(n) = \Theta(g(n))$ que $f(n)$ é assintoticamente equivalente a $g(n)$, ou que f e g possuem o mesmo crescimento assintótico, ou que f é *ensanduichada* por g, já que existem duas constantes reais não nulas c_1 e c_2 e um natural n_0 tais que

$$c_1|g(n)| \leqslant |f(n)| \leqslant c_2|g(n)| \qquad \text{para todo } n \geqslant n_o. \tag{C.17}$$

Assim, temos outra definição para o e ω:

DEFINIÇÃO C.18: Sendo duas funções $f, g\colon \mathbb{N} \to \mathbb{R}$, escrevemos $f(n) = o(g(n))$ quando f é $O(g(n))$ mas não é $\Theta(g(n))$. Analogamente, escrevemos $f(n) = \omega(g(n))$ quando f é $\Omega(g(n))$ mas não é $\Theta(g(n))$.

A equivalência entre a Definição C.18 e a Definição C.7 segue naturalmente da manipulação dos quantificadores envolvidos.

C.3 Melhor caso, pior caso e caso médio de um algoritmo

DEFINIÇÃO C.19: Sendo n um natural, o *melhor caso* de um algoritmo A é uma entrada X_{melhor} de tamanho n tal que, para qualquer outra entrada Y de tamanho n, $T_A(X_{\text{melhor}}) \leqslant T_A(Y)$. Analogamente, o *pior caso* de A é uma entrada X_{pior} de tamanho n tal que, para qualquer outra entrada Y de tamanho n, $T_A(X_{\text{pior}}) \geqslant T_A(Y)$.

Sendo k e L naturais não-nulos, tomemos como exemplo o Algoritmo C.20 (p. 156) que, dados

a) um vetor $\boldsymbol{V}[1..k]$ com k objetos representáveis cada um por L *bits* e
b) um objeto α, um dos k objetos de $\boldsymbol{V}$,

fornece como resposta o menor i tal que $\boldsymbol{V}[i] = \alpha$. Assumimos que o acesso a um determinado elemento de $\boldsymbol{V}$ pode ser feito em tempo assintoticamente constante, assim como comparações entre objetos, já que assumimos que a representação dos objetos é limitada por uma constante L. Note-se que o tamanho da entrada, em *bits*, é $(k+1)L$. No entanto, desprezamos a constante L e assumimos que o tamanho da entrada é assintoticamente equivalente a k.

O melhor caso do Algoritmo C.20 é aquele em que o laço é executado ape-

nas uma vez — quando $V[1] = \alpha$. Nesse caso, as operações executadas foram a inicialização da variável i, a comparação de i com k, a comparação de $\boldsymbol{V}[i]$ com α e a devolução de $i = 1$ como resposta. Todas essas operações podem ser realizadas em tempo assintoticamente constante no modelo RAM, uma vez assumidas nossas hipóteses acerca do tamanho das representações dos objetos e do acesso aos elementos de $\boldsymbol{V}$. Portanto, como elas são executadas uma única vez, temos que o tempo do melhor caso do Algoritmo C.20, fixados k e L, é $\Theta(1)$.

ALGORITMO C.20 — Busca linear.
ENTRADA: Um vetor $\boldsymbol{V}\{]1..k]$ e um objeto α.
SAÍDA: O menor i tal que $\boldsymbol{V}[i] = \alpha$.
1: PARA i DE 1 ATÉ k, FAÇA:
2: SE $\boldsymbol{V}[i] = \alpha$, ENTÃO,
3: DEVOLVA i,
4: FIM,
5: FIM.

Por outro lado, o pior caso do Algoritmo C.20 é aquele em que o laço é executado k vezes e α é encontrado apenas em $\boldsymbol{V}[k]$. Nesse caso, além da inicialização da variável i e da devolução de k como resposta, são executadas k comparações de i com k e k comparações de $\boldsymbol{V}[i]$ com α. Assim, temos que o tempo do pior caso do Algoritmo C.20 é $\Theta(k)$, assintoticamente equivalente a linear — ou simplesmente assintoticamente linear — no número de posições no vetor.

DEFINIÇÃO C.21: Sendo n um natural, o tempo do *caso médio* de um algoritmo A, denotado por $\mathbb{E}T_A(X)$, é o valor esperado para $T_A(X)$ tomando-se X aleatoriamente.

Em nosso exemplo, sendo A o Algoritmo C.20, assumindo-se todas as pos-

síveis entradas equiprováveis, temos, das propriedades da esperança, que

$$\mathbb{E}T_A(\boldsymbol{V}\{1,\ldots,k\},\alpha) = \sum_{i=1}^{k} \mathbb{E}\big(T_A(\boldsymbol{V}\{1,\ldots,k\},\alpha)\big|e_i\big)\mathbb{P}\big(e_i\big), \tag{C.22}$$

sendo e_i o evento

$$e_i = \big[\boldsymbol{V}[1] \neq \alpha \wedge \cdots \wedge \boldsymbol{V}[i-1] \neq \alpha \wedge \boldsymbol{V}[i] = \alpha\big]. \tag{C.23}$$

Logo,

$$\begin{aligned}\mathbb{E}T_A(\boldsymbol{V}\{1,\ldots,k\},\alpha) &= \Theta(1)\frac{1}{k} + \Theta(2)\Big(1-\frac{1}{k}\Big)\frac{1}{k} + \cdots + \Theta(k)\Big(1-\frac{1}{k}\Big)^{k-1}\frac{1}{k} \\ &= \frac{1}{k}\Theta\bigg(\sum_{i=1}^{k} i\Big(1-\frac{1}{k}\Big)^{i-1}\bigg),\end{aligned} \tag{C.24}$$

se bem que, do Lema A.44 (p. 144), para k suficientemente grande, $\sum_{i=1}^{k} i\big(1-\frac{1}{k}\big)^{i-1} \approx k^2$, o que nos traz que

$$\mathbb{E}T_A(\boldsymbol{V}\{1,\ldots,k\},\alpha) = \frac{1}{k}\Theta(k^2) = \Theta(k). \tag{C.25}$$

Dessarte, concluímos que o tempo do caso médio para o algoritmo da busca linear (Algoritmo C.20) é também assintoticamente linear no número de posições do vetor, a despeito de não ser essa a complexidade de tempo do melhor caso.

D ALGUMAS DEFINIÇÕES SOBRE HIPERGRAFOS

Apresentamos a seguir algumas definições concernentes a hipergrafos, necessárias para a família de esquemas de *hashing* apresentadas na Seção 2.4 (p. 80).

DEFINIÇÃO D.1: Um *hipergrafo* G é uma estrutura $(V(G), E(G))$ composta por um conjunto finito qualquer $V(G)$ de elementos chamados *vértices* e por um conjunto $E(G)$ de arestas, sendo cada aresta um subconjunto não-vazio de $V(G)$.

Fica evidente que, num hipergrafo G qualquer, $E(G) \subseteq 2^{V(G)} \setminus \{\emptyset\}$. Usamos 2^A para denotar o *conjunto das partes* de A — o conjunto de todos os subconjuntos de A. Segue-se, então, o resultado do Teorema D.2.

TEOREMA D.2: *O número máximo de arestas que um hipergrafo com n vértices pode ter é $2^n - 1$.*

Demonstração: Todo conjunto finito A possui $2^{|A|}$ subconjuntos, como mostrado, por exemplo, em Graham, Knuth e Patashnik (1994). Como $|V(G)| = n$, e como $E(G) \subseteq 2^{V(G)} \setminus \{\emptyset\}$, $|E(G)| \leqslant 2^n - 1$. □

Costumamos representar hipergrafos por diagramas de Venn. Cada vértice é identificado por um ponto, e cada aresta, por uma área finita delimitada no diagrama por um perímetro fechado. Por exemplo, a Figura D.4 (p. 159) representa o hipergrafo cujo conjunto de vértices é o conjunto $V(G) = \{1, 2, 3, 4, 5\}$ e cujo conjunto de arestas é o conjunto

$$E(G) = \{\{1, 2\}, \{3, 4, 5\}, \{2, 3, 4\}, \{1, 2, 3, 5\}, \{5\}\}. \quad \text{(D.3)}$$

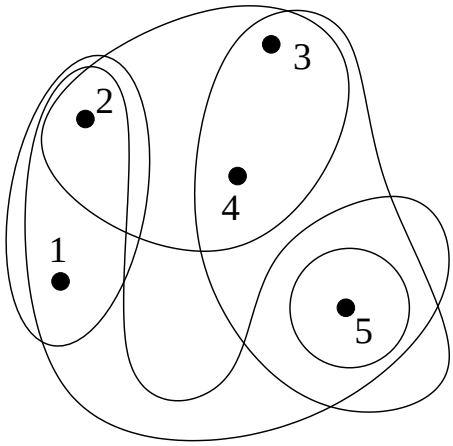


FIGURA D.4 — Um hipergrafo representado por um diagrama de Venn.

Outra maneira de se representarem hipergrafos é por diagramas de estrelas. Nesse tipo de diagrama, cada vértice é identificado por um ponto, e cada aresta, por uma *estrela* — uma união de segmentos de curva num só ponto central tendo em cada extremidade um vértice. Na Figura D.6 (p. 160) representamos o hipergrafo cujo conjunto de vértices é o conjunto $V(G) = \{1, 2, 3, 4, 5\}$ e cujo conjunto de arestas é o conjunto

$$E(G) = \{\{1, 2\}, \{3, 4, 5\}, \{2, 3, 4\}, \{1, 2, 3, 5\}\}. \tag{D.5}$$

Note-se que com diagramas de estrelas não conseguimos representar arestas unitárias.

DEFINIÇÃO D.7: Sendo r um natural qualquer, quando todas as arestas dum hipergrafo G_r possuem exatamente r vértices, dizemos que G_r é um r-hipergrafo.

Fica evidente que, num r-hipergrafo G qualquer, $E(G) \subseteq \binom{V(G)}{r}$. Usamos $\binom{A}{r}$ para denotar o *conjunto das combinações* de n elementos de A — o conjunto de todos os subconjuntos de A que têm n elementos (Notação A.1). Segue-se, então, o resultado do Teorema D.8.

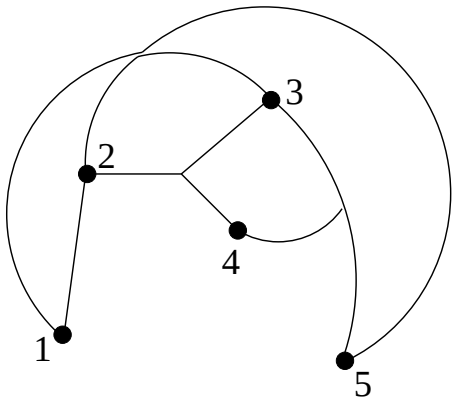


FIGURA D.6 — Um hipergrafo representado por um diagrama de estrelas.

TEOREMA D.8: *O número máximo de arestas que um r-hipergrafo com n vértices pode ter é $\binom{n}{r}$.*

Demonstração: Todo conjunto finito A possui $\binom{|A|}{r}$ subconjuntos de cardinalidade r, como mostrado, por exemplo, em Graham, Knuth e Patashnik (1994). Como $|V(G)| = n$, e como $E(G) \subseteq \binom{V(G)}{r}$, $|E(G)| \leqslant \binom{n}{r}$. □

Note-se que chamamos os 2-hipergrafos simplesmente de grafos.

DEFINIÇÃO D.9: Sendo $k \geqslant 2$ um inteiro, dizemos que um hipergrafo G é k-partido se $V(G)$ pode ser escrito como a união de k conjuntos não-vazios e disjuntos entre si, $V(G) = V_1 \cup \cdots \cup V_k$, e se, para toda aresta $e \in E(G)$, não há dois vértices em e que pertençam ao mesmo V_j, para cada $j \in \{1, \ldots, k\}$.

TEOREMA D.10: *Se G é k-partido, então, toda aresta de G tem no máximo k vértices.*

Demonstração: Se alguma aresta de G tem mais de k vértices, então, do Princípio da Casa dos Pombos (como exposto, por exemplo, em Graham, Knuth e Patashnik (1994,

cap. 4)), há ao menos 2 vértices que pertencem ao mesmo V_j, para $j \in \{1, \ldots, k\}$. □

Mostramos na Figura D.11 um hipergrafo 4-partido, em que $V(G) = \{1, \ldots, 16\} = V_1 \cup V_2 \cup V_3 \cup V_4$, sendo $V_1 = \{1, 2, 3, 4\}$, $V_2 = \{5, 6, 7, 8\}$, $V_3 = \{9, 10, 11, 12\}$ e $V_4 = \{13, 14, 15, 16\}$.

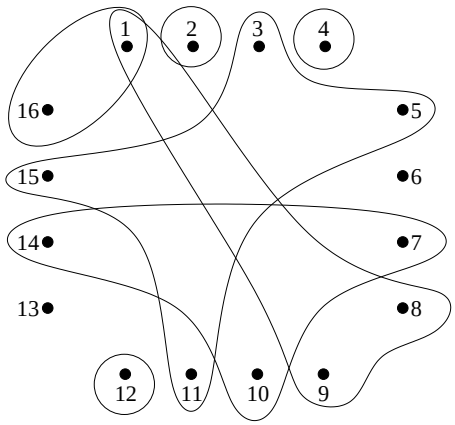


FIGURA D.11 — Exemplo de um hipergrafo 4-partido.

Observemos que, num k-hipergrafo k-partido G em que $V(G)$ é a união disjunta dos conjuntos não-vazios $V_1, \ldots, V_k$, cada aresta com k vértices de G pode ser considerada uma sequência $(u_1, \ldots, u_k)$ de vértices, sendo que $u_1 \in V_1, \ldots, u_k \in V_k$. Dessa asserção, temos o resultado do Teorema D.12.

TEOREMA D.12: *Nenhum k-hipergrafo k-partido G com n vértices pode ter mais que $\left(\frac{n}{k}\right)^k$ arestas.*

Demonstração: O número máximo de arestas de G é dado por $|V_1| \cdots |V_k|$. Se os fatores desse produto pudessem ser reais, esse produto atingiria seu máximo quando $|V_j| = \frac{n}{k}$ para todo $j \in \{1, \ldots, k\}$. Ora, nesse caso, o produto valeria $\left(\frac{n}{k}\right)^k$. □

Mostramos na Figura D.13 um 3-hipergrafo 3-partido, em que $V(G) = \{1, \ldots, 6\} = V_1 \cup V_2 \cup V_3$, sendo $V_1 = \{1\}$, $V_2 = \{2, 3\}$ e $V_3 = \{4, 5, 6\}$.

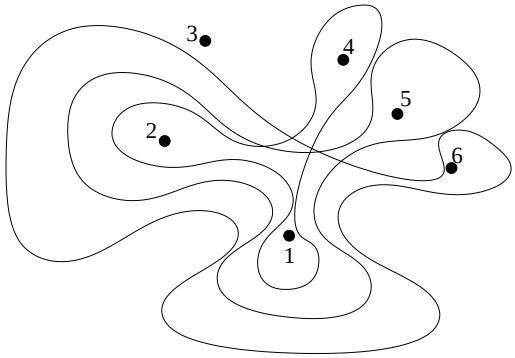


FIGURA D.13 — Exemplo de um 3-hipergrafo 3-partido.

Podemos estender a hipergrafos todos os conceitos pertinentes ao conceito de *adjacência* que temos em grafos comuns.

DEFINIÇÃO D.14: Num hipergrafo qualquer, dizemos que dois vértices u_1 s u_2 são *adjacentes* se existe ao menos uma aresta e tal que $u_1 \in e$ e $u_2 \in e$. Duas arestas e_1 e e_2 são *adjacentes* se $e_1 \cap e_2 \neq \emptyset$. Se um vértice v pertence a uma aresta e, dizemos que e é *incidente em* v, ou *incidente sobre* v. O *grau* de um vértice é o número de arestas que incidem nele.

Por exemplo, no hipergrafo representado pela Figura D.11 (p. 161), o vértice 5 é adjacente ao vértice 15, mas não é ao vértice 14. As arestas $\{1, 8, 9\}$ e $\{1, 16\}$ são adjacentes, mas as arestas $\{7, 10, 14\}$ e $\{3, 5, 11, 15\}$ não são. Os vértices 13 e 6 têm grau 0, e o vértice 1 tem grau 2. Todos os demais possuem grau 1.

DEFINIÇÃO D.15: Num hipergrafo, uma *cadeia* de tamanho k é uma sequência de

vértices intercalados com arestas

$$u_1, e_1, u_2, e_2, \ldots, u_{k-1}, e_{k-1}, u_k \tag{D.16}$$

tal que:

a) $u_i \neq u_j$ para todo i e todo j distintos em $\{1, \ldots, k-1\}$;

b) $e_i \neq e_j$ para todo i e todo j distintos em $\{1, \ldots, k-1\}$;

c) para todo $j \in \{1, \ldots, k-1\}$, $u_j, u_{j+1} \in e_j$.

Quanto $u_1 \neq u_k$, dizemos que a cadeia é um *caminho* de u_1 a u_k. Quando $u_1 = u_k$, dizemos que a cadeia é um *ciclo*. Um hipergrafo G é dito *conexo* quando existe caminho entre quaisquer dois vértices distintos de G e dito *desconexo* caso contrário. Um hipergrafo G é dito *cíclico* quando possui ao menos um ciclo e dito *acíclico* caso contrário.

No hipergrafo da Figura D.11, um caminho do vértice 16 ao vértice 9 pode ser a cadeia

$$16, \{1, 16\}, 1, \{1, 8, 9\}, 9. \tag{D.17}$$

Notemos que esse hipergrafo é acíclico. No hipergrafo da Figura D.4 (p. 159), um exemplo de ciclo é a cadeia

$$3, \{3, 4, 5\}, 5, \{1, 2, 5\}, 1, \{1, 2\}, 2, \{2, 3, 4\}, 3. \tag{D.18}$$

TEOREMA D.19: *Todo hipergrafo G acíclico que não possui arestas unitárias tem ao menos um vértice com grau menor que 2.*

Demonstração: Seja G um hipergrafo acíclico sem arestas unitárias. Se G tem 1 vértice, G é acíclico trivialmente. Se G tem 2 vértices, v_1 e v_2, o conjunto $E(G)$ só pode

ser vazio ou o conjunto $\{\{1, 2\}\}$. Mas G é acíclico em ambos os casos. Assumamos, então, que G tem $t > 2$ vértices.

Suponhamos, por contradição, que todo vértice de G tem grau no mínimo 2. Tomemos um vértice v_1 qualquer de G. Como v_1 tem grau no mínimo 2, existem ao menos duas arestas distintas e_0 e e_1 incidentes em v_1. Já que e_1 não é unitária, e_1 é também incidente num $v_2 \neq v_1$. Não obstante, por ser no mínimo 2 o grau de v_2 e por ser G acíclico, existe uma aresta $e_2 \neq e_1$ incidente a um v_3 distinto de v_1 e de v_2. Indutivamente, conseguimos um caminho

$$v_1, e_1, v_2, \ldots, e_{t-1}, v_t \tag{D.20}$$

em que $e_j \neq e_0$ para todo $j \in \{1, \ldots, t-1\}$, pois, se e_0 fosse igual a algum e_j,

$$v_1, \ldots, e_j, v_1 \tag{D.21}$$

seria um ciclo. Porém, e_0 é incidente a algum v_j para algum $j \in \{2, \ldots, t\}$, pois G não possui arestas unitárias, e, consequentemente, a cadeia

$$v_1, \ldots, e_{j-1}, v_j, e_0, v_1 \tag{D.22}$$

é um ciclo, um absurdo. □

E ESQUEMAS DINÂMICOS DE *HASHING*

Além de outras muitas aplicações, frequentemente usamos *Hashing* para modelar e implementar estruturas de dados — estruturas que nos permitem armazenar e gerenciar dados num espaço de memória dum modelo computacional. Quando temos uma determinada coleção de dados S, podemos tomar S como um conjunto de chaves, subconjunto dum universo U, e, assim, usar algum dos esquemas apresentados nos Capítulos 2 e 3 para construir uma *hash* perfeita e mínima para S, mapeando as chaves para endereços que podem ser considerados endereços dos espaços de memória onde serão armazenados os dados.

Usar um esquema de *hashing* perfeito, mínimo, prático e eficiente em tempo e espaço para construir uma estrutura de dados é ótimo se queremos para nossa estrutura apenas as operações de construir — ou inicializar — a estrutura e buscar por um dado. Afinal, assumindo limitado por uma constante o tamanho da representação dos dados, construir uma estrutura com n dados levaria tempo $O(n)$, e buscar por um dado levaria tempo $O(1)$. Note-se que ambas as complexidades são ótimas, na medida em que é fácil verificar que construir uma estrutura com n dados não pode ser feito em tempo $o(n)$. Buscar por um dado em tempo $O(1)$, além de ser ótimo, é muito melhor que o que conseguimos fazer com muitas das clássicas estruturas de dados baseadas em árvores (apresentadas, por exemplo, por Cormen, Leiserson e Rivest (1990, cap. 12–14)), as quais realizam buscas em tempo $\Theta(\log n)$.

No entanto, estruturas de dados geralmente requerem outras operações, como inserção e deleção de dados, por exemplo. Se construirmos uma *hash* h perfeita para um determinado conjunto de chaves S, seguramente h deixará de ser perfeita se adicionarmos uma chave a S. Analogamente, se construirmos h mínima para S, seguramente h deixará de ser mínima após uma chave ser removida de S. Assim, a

cada inserção e deleção de chave, se ainda queremos uma *hash* perfeita ou mínima, podemos precisar recalcular h. Uma abordagem seria executar algum dos esquemas eficientes em tempo que apresentamos nos Capítulos 2 ou 3, os quais chamamos de esquemas de *hashing estáticos*. No entanto, isso requereria um tempo $\Omega(n)$ a cada inserção ou deleção, conforme o Teorema 1.16 (p. 31), uma complexidade de tempo assintoticamente muito além do tempo $\Theta(\log n)$ próprio das estruturas clássicas baseadas em árvores.

Quando executamos um esquema de *hashing* estático sobre um novo conjunto de chaves S', obtido a partir duma operação de inserção ou de deleção de uma chave de um antigo S, simplesmente desconsideramos a *hash* que já havíamos calculado para S. Conforme Czech, Havasb e Majewski (1997, cap. 7), um *esquema dinâmico de hashing* consiste, não obstante:

a) num procedimento de *inicialização* ou *construção*, que utiliza um esquema de *hashing* estático para computar uma *hash* h sobre um universo U para um conjunto de chaves S;
b) num procedimento de *inserção*, que recebe um conjunto S, uma *hash* para S e uma chave $x \notin S$ e devolve uma *hash* para $S \cup \{x\}$.
c) num procedimento de *deleção*, que recebe um conjunto S, uma *hash* para S e uma chave $x \in S$ e devolve uma *hash* para $S \setminus \{x\}$.
d) num procedimento de *busca*, que recebe uma chave x e uma *hash* h e computa o endereço $h(x)$.

Dizemos que um esquema dinâmico é perfeito quando, após uma operação de inicialização, de inserção ou de deleção, a *hash* devolvida sempre é perfeita. De igual modo, quando, após qualquer uma dessas operações, a *hash* sempre é mínima, dizemos que o esquema é mínimo.

Aho e Lee (1986) foram os primeiros a propor um esquema dinâmico perfeito para o qual as operações de inserção e de deleção gozam de tempo $O(1)$. No entanto,

para que essas operações possam de fato ser executadas em tempo $O(1)$, exige-se a restrição de que a chave a ser inserida ou removida seja tomada do espaço de todas as possíveis chaves com distribuição uniforme. Propondo um esquema dinâmico perfeito baseado no FKS, Dietzfelbinger et al. (1988) alcançaram os mesmos resultados para os tempos de inserção e de deleção para qualquer distribuição de probabilidade sobre o espaço das chaves a serem inseridas ou removidas.

Dietzfelbinger et al. (1988) também mostraram que complexidades de tempo $O(1)$ para as operações de inserção e de deleção só são possíveis para esquemas dinâmicos aleatorizados. Ademais, mostraram que $\Omega(\log n)$ é uma cota inferior para essas operações em esquemas determinísticos que usam espaço linear no número de chaves. Aqui há de se fazer uma observação importante. Embora precisemos ter esquemas dinâmicos aleatorizados se queremos tempo constante para as operações de inserção e deleção, isso não significa que os esquemas estáticos nos quais nos baseamos para construirmos os dinâmicos devam igualmente ser aleatorizados. Como exposto por Czech, Havasb e Majewski (1997), o esquema dinâmico baseado no FKS, por exemplo, é aleatorizado, mesmo o FKS sendo determinístico.

Muitos outros esquemas de *hashing* dinâmicos e perfeitos já foram desenvolvidos, a exemplo dos resultados de Brodnik e Munro (1999) e de Pagh e Rodler (2004). Nenhum ainda, no entanto, que se inspirasse nos excelentes esquemas estáticos de Botelho, Kohayakawa e Ziviani (2005) e de Botelho, Pagh e Ziviani (2007). Acreditamos que nossa abordagem determinística desses esquemas, apresentadas do Capítulo 3, poderá contribuir para a construção de esquemas de *hashing* dinâmicos, perfeitos, mínimos, práticos e eficientes em tempo e em espaço.

ÍNDICE ALFABÉTICO