Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo Comparativo do Uso de Hashing Perfeito Mínimo

Fabiano Cupertino Botelho

Belo Horizonte
26 de novembro de 2004

Fabiano Cupertino Botelho

Estudo Comparativo do Uso de Hashing Perfeito Mínimo

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte
26 de novembro de 2004

Resumo

Uma função *hash* perfeita mínima é uma função bijetora que mapeia um conjunto estático de n chaves em uma tabela *hash* de tamanho n. Uma vantagem das funções *hash* perfeitas mínimas em termos de economia de espaço é que não há necessidade de armazenar as chaves, apenas a função é suficiente para calcular uma entrada na tabela. Esta dissertação apresenta um estudo comparativo dos principais algoritmos para gerar funções *hash* perfeitas mínimas disponíveis na literatura. Além disso, comparamos funções *hash* perfeitas mínimas com o tradicional método endereçamento aberto que trata colisões por *hashing* linear. A taxa de ocupação ou fator de carga é definida pela razão entre o número de registros armazenados na tabela e o seu tamanho. Um resultado interessante deste estudo é que a avaliação da função *hash* perfeita mínima para encontrar uma posição da tabela *hash* é mais rápida do que o método endereçamento aberto para ocupação da tabela acima de 40%. Assim, para que o método endereçamento aberto tenha desempenho melhor do que funções *hash* perfeitas mínimas é necessário manter pelo menos 60% das entradas vazias.

Abstract

A minimal perfect hash function is a bijection function that maps a static set of n keys into a hash table of size n. An advantage of minimal perfect hash functions in terms of space is that there is no need to store the set of keys, just the function is sufficient to calculate the hash table entry. This thesis presents a comparative study of the main algorithms in the literature for generating minimal perfect hash functions. Moreover, we compare minimal perfect hash functions with the traditional open addressing method using linear hashing to resolve collisions. The load factor is defined as the ratio between the number of records stored in the table and its size. An interesting result of this study is that the time to calculate a hash table entry by the minimal perfect hash function is smaller than the time for the open addressing method when the load factor is above 40%. Thus, the open addressing method is faster than minimal perfect hash functions only when more than 60% of the hash table entries are kept empty.

Agradecimentos

Em primeiro lugar gostaria de agradecer a Deus por ter concedido a mim vida e sabedoria para realizar um sonho de infância.

Gostaria de agradecer aos meus queridos pais Maria Lúcia de Lima Botelho e José Vitor Botelho pelos sacrifícios realizados para esta conquista. Gostaria de agradecer também as minhas queridas irmãs Cristiane Cupertino Botelho e Gleiciane Cupertino Botelho, pelo carinho e amor das duas melhores irmãs do mundo.

Também gostaria de agradecer a minha noiva e futura esposa Janaína Marcon Machado pelo amor, compreensão, companheirismo e incentivo durante momentos nos quais tive vontade de desistir de tudo. Obrigado Jana, com a graça de Deus em nossas vidas continuaremos a ser muito felizes.

Jamais poderia esquecer de agradecer aos meus queridos tios Sudário Alves e Márcia Novaes Alves, os quais me acolheram com todo carinho e foram verdadeiros pais para mim durante os 17 meses de mestrado.

Gostaria de agradecer ao meu orientador Prof. Nivio Ziviani, por ter sido muito mais do que um orientador e por ter dado a mim a oportunidade de aprender um pouquinho com a sua experiência.

Gostaria de agradecer aos amigos do Laboratório para Tratamento da Informação Anísio Mendes Lacerda, Álvaro Rodrigues Pereira Júnior, Charles Ornelas Almeida, Claudine Santos Badue, Fábio Cossenzo, Marco Antônio Pinheiro de Cristo, Pável Calado, Sávio Ornelas Almeida e Thierson Couto Rosa pela ajuda com a defesa e com o desenvolvimento da dissertação. Também gostaria de agradecer ao amigo David Menoti pelas discussões e críticas que enriqueceram muito esta dissertação. Também gostaria de agradecer aos meus amigos de república Ligiane Alves de Souza, Luciano Flávio Meira e Marcos Vinícius Teixeira por terem me ajudado a superar com bom humor momentos difíceis que passei durante o mestrado.

Gostaria de agradecer também a meus amigos de graduação e também companheiros de mestrado Wagner Ferreira de Barros e Fabrício Orlando Damasceno por compartilharem comigo as dificuldades e alegrias pelas quais passamos durante o mestrado.

Por fim, gostaria de agradecer aos membros da banca Prof. Edleno Silva de Moura e Prof. Wagner Meira Júnior pelas críticas que ajudaram a melhorar o trabalho.

Sumário

Lista de Figuras ix

Lista de Tabelas xi

1 Introdução 1

1.1 Motivação 1

1.2 Trabalhos Relacionados 3

1.3 Objetivos 5

1.4 Contribuições 6

1.5 Conceitos Básicos e Notação 6

1.6 Estrutura da Dissertação 9

2 Funções *Hash* 11

2.1 Função *Hash* Universal 11

2.2 Função *Hash* de Zobrist 13

2.3 Função *Hash* de Jenkins 15

2.4 Armazenamento das Funções em Memória Externa 20

3 Algoritmos Avaliados 21

3.1 Estrutura da Tabela *Hash* 21

3.2 Endereçamento Aberto com *Hashing* Linear 22

3.3 A Família de Algoritmos MWHC-r 23

3.3.1 Mapeamento e Ordenação 24

3.3.2 Pesquisa 27

3.3.3 Implementação 28

3.4 O Algoritmo FCH 34

3.4.1 Mapeamento 35

3.4.2 Ordenação 36
3.4.3 Pesquisa 38

4 Resultados Experimentais 43
4.1 Conjunto de Chaves 44
4.2 Impacto das Funções *Hash* no Método Endereçamento Aberto 44
4.3 Hashing Perfeito Versus Hashing Linear 47
4.4 Geração da Função Hashing Perfeita Mínima 51
4.4.1 Impacto das Funções *Hash* no Tempo de Geração das FHPMs . . . 52
4.4.2 Experimentos Sobre a Complexidade dos Algoritmos 54

5 Conclusões e Trabalhos Futuros 59

Bibliografia 62

Lista de Figuras

1.1 (a) Função *hash* perfeita. (b) Função *hash* perfeita mínima. 7

2.1 Geração do conjunto de pesos P da função *hash* universal. 12
2.2 Implementação da função *hash* universal. 13
2.3 Geração do conjunto de pesos P da função hz. 14
2.4 Implementação da função hz. 15
2.5 Modelagem da função hj. 16
2.6 Implementação do embaralhamento dos *bits* do estado interno. 17
2.7 Implementação da função hj. 19
2.8 Geração da semente para o gerador de números pseudo-aleatórios. 20

3.1 Estrutura de dados que representa a tabela *hash*. 22
3.2 Endereçamento aberto 23
3.3 Teste de aciclicidade 25
3.4 Pseudocódigo para o teste de aciclicidade de um r-grafo G_r. 26
3.5 Pseudocódigo para os passos de mapeamento e ordenação. 28
3.6 Passo de pesquisa para um 2-grafo com 6 vértices e 5 arestas. 29
3.7 Pseudocódigo para o passo de pesquisa. 29
3.8 (a) 2-grafo com 6 vértices e 5 arestas. (b) Representação do 2-grafo. 30
3.9 Descrição da estrutura de dados para representar um r-grafo. 31
3.10 Funções para criar um r-grafo vazio e para liberar a memória alocada por um r-grafo. 31
3.11 Processo de criação de um 2-grafo com 4 vértices e 2 arestas. 32
3.12 Função para inserir uma aresta a no r-grafo. 33
3.13 Função para apagar uma aresta a do r-grafo. 33
3.14 (a) Passo de mapeamento. (b) Passos de ordenação e pesquisa. 35
3.15 Pseudocódigo para o passo de mapeamento. 37

3.16 Estrutura de dados auxiliar para o passo de ordenação. 37
3.17 Pseudocódigo para o passo de ordenação. 38
3.18 Estrutura de índice para o preenchimento de uma posição vazia da tabela *hash*. 40
3.19 Pseudocódigo para o passo de pesquisa. 41

4.1 Tempo de execução dos algoritmos utilizando as funções hu e hz. 54
4.2 Tempo de execução dos algoritmos utilizando a função *hash* de Jenkins. . . 55
4.3 Distribuição do tamanho dos *Buckets* para as funções *hash* de Zobrist e Jenkins. 57
4.4 Tamanho do maior *Bucket*, $O(\log n)$. 57

Lista de Tabelas

2.1 Inicialização do estado interno. 16

2.2 Bloco de texto a ser combinado com o estado interno. 17

2.3 Combinação do bloco de texto com o estado interno através de adições. . . 18

4.1 Simbologia e abreviações utilizadas na apresentação dos resultados. 43

4.2 Propriedades dos conjuntos de chaves utilizados nos experimentos. 44

4.3 Influência do fator de carga no tempo para pesquisar todas as chaves do conjunto extraído da coleção TREC-VLC2 considerando as funções hu, hz e hj. 45

4.4 Influência do fator de carga no tempo para pesquisar todas as chaves do conjunto extraído da coleção de URLs considerando as funções hu, hz e hj. 46

4.5 Comparação da utilização de FHPMs versus *hashing* linear considerando o tempo para pesquisar todas as chaves do conjunto extraído da coleção TREC-VLC2, variando as funções hu, hz e hj na composição das FHPMs e do *hashing* linear. 48

4.6 Comparação da utilização de FHPMs versus *hashing* linear considerando o *overhead* de espaço para armazenar a tabela *hash* e o arranjo g de cada FHPM para a coleção TREC-VLC2. 49

4.7 Comparação da utilização de FHPMs versus *hashing* linear considerando o tempo para pesquisar todas as URLs, variando as funções hu, hz e hj na composição das FHPMs e do *hashing* linear. 50

4.8 Comparação da utilização de FHPMs versus *hashing* linear considerando o *overhead* de espaço para armazenar a tabela *hash* e o arranjo g de cada FHPM para a coleção de URLs. 50

4.9 Tempo de geração de FHPMs para o conjunto de chaves extraído da coleção TREC-VLC2 utilizando os algoritmos MWHC-2 e MWHC-3, variando as funções hu, hz e hj na composição das FHPMs. 52

4.10 Tempo de geração de FHPMs para o conjunto de chaves extraído da coleção TREC-VLC2 utilizando o algoritmo FCH. 53
4.11 Tempo de geração de FHPMs para a coleção de URLs utilizando os algoritmos MWHC-2 e MWHC-3, variando as funções hu, hz e hj na composição das FHPMs. 53
4.12 Tempo de geração de FHPMs para a coleção de URLs utilizando o algoritmo FCH. 54
4.13 Impacto das funções hu e hz no tempo de execução do algoritmo MWHC-2. 55
4.14 Impacto das funções hj e hu no tempo de execução dos algoritmos MWHC-2 e MWHC-3, respectivamente. 55
4.15 Impacto das funções hz e hj no tempo de execução do algoritmo MWHC-3. 56
4.16 Impacto das funções hu e hz no tempo de execução do algoritmo FCH. . . 56
4.17 Impacto da função hj no tempo de execução do algoritmo FCH. 56

Capítulo 1

Introdução

1.1 Motivação

Seja S um conjunto de n chaves distintas pertencentes a um universo finito U de chaves. As chaves em S são armazenadas de forma que consultas perguntando se uma chave $x \in U$ está em S possam ser respondidas. Este problema é chamado **problema do dicionário**. Várias abordagens para o problema do dicionário têm sido exploradas. Uma delas é computar uma função $h(x)$ para determinar a localização de uma chave em uma tabela, levando a uma classe muito eficiente de métodos de pesquisa conhecida como **métodos de *hashing*** [30]. A função h é chamada **função *hash*** e a tabela endereçada por ela é chamada **tabela *hash***.

Como descrito em Ziviani [30, Capítulo 5], existem dois problemas relacionados com os métodos de *hashing*. O primeiro consiste em obter uma função *hash* que distribua os registros de forma uniforme entre as entradas da tabela. O segundo ocorre quando duas chaves distintas são mapeadas no mesmo endereço da tabela, o que caracteriza uma **colisão**.

Por melhor que seja a função *hash*, existe uma alta probabilidade de haver colisões. Isto pode ser ilustrado pelo paradoxo do aniversário [10, 30], o qual diz que em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que 2 pessoas comemorem aniversário no mesmo dia. Isso significa que a probabilidade de ocorrer colisões é maior do que 50% quando uma função *hash* uniforme é utilizada para endereçar 23 chaves randômicas em uma tabela de tamanho 365. Dessa forma, é preciso estabelecer mecanismos para resolver as colisões ou então impedir que elas aconteçam.

Existem vários métodos para armazenar n registros em uma tabela de tamanho $m \geq n$, os quais utilizam os endereços vazios na própria tabela para resolver as colisões. Tais métodos são chamados **endereçamento aberto** (***open addressing***) [30]. Estes métodos envolvem uma certa quantidade de espaço desperdiçado devido a localizações não utilizadas na tabela *hash*. Além disso, existe também um desperdício de tempo para resolver colisões quando duas chaves são mapeadas no mesmo endereço da tabela *hash*.

Para conjuntos estáticos, é possível computar uma função $h(x)$ para encontrar qualquer chave na tabela em uma única tentativa, sem a ocorrência de colisões. Esta função é chamada **função *hash* perfeita**. Uma função *hash* perfeita que preserva a ordem previamente estabelecida entre as chaves é chamada **função *hash* perfeita com ordem preservada**. Uma função *hash* perfeita que mapeia um conjunto de chaves de tamanho n em endereços de uma tabela *hash* de igual tamanho é chamada **função *hash* perfeita mínima**. Uma função *hash* perfeita mínima pode evitar totalmente o problema de desperdício de espaço e de tempo.

Funções *hash* perfeitas mínimas são utilizadas para permitir armazenamento e recuperação eficiente de itens provenientes de conjuntos estáticos, tais como palavras em linguagem natural, palavras reservadas em linguagens de programação ou sistemas interativos, URLs (*Universal Resource Locations*) nas máquinas de busca, conjuntos de itens freqüentes em técnicas de mineração de dados. Conseqüentemente, existem aplicações para funções *hash* perfeitas mínimas em sistemas de recuperação de informação, sistemas de bancos de dados, hipertexto, hipermídia, sistemas de tradução de linguagens, sistemas de comércio eletrônico, compiladores, sistemas operacionais, entre outras.

Gerar funções *hash* perfeitas, especialmente para grandes conjuntos, pode não ser fácil já que estas funções são muito raras. De acordo com Knuth [21], o número total de funções *hash* para mapear um conjunto S com n chaves para o intervalo $[0, m-1]$ é m^n, sendo que somente $m(m-1)\dots(m-n+1)$ são perfeitas. Assim, a probabilidade de que não haja colisão é a razão $(m(m-1)\dots(m-n+1))/m^n$, a qual tende para zero muito rápido. Para $m = 13$ e $n = 10$, a probabilidade de que não haja colisões é 0.0074. Para funções *hash* perfeitas mínimas, onde $m = n$, a probabilidade é ainda menor. Neste caso temos que a probabilidade é dada por $n!/n^n$. Para $n = 10$, a probabilidade de que não haja colisões é somente 0.00036.

Muitos métodos para gerar uma função *hash* perfeita mínima usam a abordagem MOS, Mapeamento (*Mapping*), Ordenação (*Ordering*) e Pesquisa (*Searching*), uma descrição cunhada por Fox et al. (cf. [16, 13]). Na abordagem MOS, a construção da função *hash* perfeita mínima é desempenhada em três passos. Primeiro, o passo de mapeamento trans-

forma o conjunto de chaves do universo original para um novo universo. Segundo, o passo de ordenação estabelece a ordem na qual valores *hash* serão atribuídos para as chaves. Terceiro, o passo de pesquisa tenta atribuir valores *hash* para as chaves.

O objetivo desta dissertação é apresentar um estudo comparativo de funções *hash* perfeitas mínimas para resolver o problema do dicionário. Mais especificamente, objetivamos comparar funções *hash* perfeitas mínimas com o tradicional método de endereçamento aberto que trata colisão por *hashing* linear [21, 30]. O intuito é estabelecer os compromissos existentes. Além disso, visamos identificar e comparar os melhores algoritmos conhecidos para gerar funções *hash* perfeitas mínimas.

1.2 Trabalhos Relacionados

Métodos de *hashing* têm sido um tópico de estudo muito explorado. Isto se deve a grande quantidade de aplicações onde eles podem ser utilizados. Em [26], Özel e Güvenir utilizam funções *hash* perfeitas para minerar regras de associação a partir de transações de bancos de dados. Recentemente, funções *hash* perfeitas mínimas têm sido utilizadas para mapear um conjunto de URLs (*Universal Resource Locations*) para o conjunto de vértices de um grafo da *Web* [3]. Fox, Chen e Heath [14] utilizam funções *hash* perfeitas mínimas para assegurar eficiência de espaço e tempo no sistema de indexação de um banco de dados desenvolvido especificamente para aplicações em recuperação de informação. Um exemplo de dicionário é o vocabulário de um arquivo invertido [1, 29]. Neubert [25] utilizou funções *hash* perfeitas mínimas para representar o vocabulário de um arquivo invertido construído de forma distribuída.

Através da construção de um algoritmo, Jaeschke [19] provou que é sempre possível gerar uma função *hash* perfeita mínima para um dado conjunto de chaves. O principal problema com o algoritmo de Jaeschke [19] é a sua complexidade exponencial no número de chaves para gerar a função.

Existem vários algoritmos disponíveis na literatura para computar uma função *hash* perfeita mínima, alguns exponenciais no tamanho do conjunto de chaves [5, 19, 28], outros polinomiais [4, 7, 6, 27]. Czech, Havas e Majewski [8] apresentam um amplo *survey* sobre métodos de *hashing* perfeito. Eles revisam os resultados teóricos e práticos mais importantes publicados na literatura. Mehlhorn [23] mostrou que o limite inferior de espaço para armazenar uma função *hash* perfeita é $\Omega(n)$ *bits* (ou $\Omega(n/\log_2 n)$ palavras de computador de tamanho $\log n$), onde n é o tamanho do conjunto de chaves. Fox et al. [12] mostra-

ram que o limite inferior de espaço para armazenar uma função *hash* perfeita com ordem preservada é $\Omega(n \log_2 n)$ *bits* (ou $\Omega(n)$ palavras de computador de tamanho $\log n$).

Majewski, Wormald, Havas e Czech [22] generalizam o trabalho inicialmente apresentado em [7] e propõem uma família de algoritmos práticos e eficientes para gerar funções *hash* perfeitas mínimas com ordem preservada. Os algoritmos são baseados na abordagem MOS. No passo de mapeamento (*mapping*), um grafo não direcionado randômico[1] é gerado a partir do conjunto de chaves. No passo de ordenação (*ordering*), o endereço na tabela *hash* onde uma chave deve estar localizada é associado com cada aresta. No passo de pesquisa (*searching*), valores são atribuídos aos vértices do grafo para gerar a função.

A família de algoritmos em [22, 7] trabalha sobre grafos acíclicos para gerar as funções *hash* perfeitas mínimas com ordem preservada. A geração de grafos randômicos acíclicos exige que o número de vértices do grafo seja pelo menos 2.09 vezes o número de arestas. Como conseqüência, mais espaço é necessário para armazenar a função. Para diminuir o espaço de armazenamento da função, Majewski et al. [22] propõem a utilização de hipergrafos (ou r-grafos)[2]. Neste caso, o número de vértices do grafo deve ser pelo menos 1.23 vezes o número de arestas. Embora o espaço de armazenamento da função seja menor, o seu tempo de avaliação é maior. A família de algoritmos é discutida em mais detalhes na Seção 3.3, sendo referenciada pelo nome MWHC-r, onde o MWHC é devido à seus autores e o r é proveniente do fato dos algoritmos utilizarem r-grafos para gerar as funções.

Os algoritmos propostos por Fox et al. [15] e por Czech e Majewski [6] também se baseiam na abordagem MOS para gerar funções *hash* perfeitas mínimas. Tais algoritmos geram um grafo bipartido no passo de mapeamento, o qual pode conter ciclos. Em [15], são apresentados dois algoritmos que necessitam de $\log n$ e 10 *bits* por chave para armazenar a função gerada, respectivamente. Fox et al. [15] mostraram experimentos sugerindo uma complexidade de tempo linear no tamanho do conjunto de chaves para gerar a função. No entanto, Czech, Havas e Majewski [8, Seção 6.7] mostraram que a complexidade de tempo dos algoritmos é exponencial no pior caso. Já o algoritmo proposto por Czech e Majewski [6] é linear, porém exige aproximadamente $2 \log n$ *bits* por chave para armazenar a função gerada.

O melhor algoritmo disponível na literatura em termos de eficiência de espaço para armazenar uma função *hash* perfeita mínima foi proposto por Fox, Chen e Heath [14]. O algoritmo também é baseado na abordagem MOS. Diferentemente dos anteriores, o algoritmo não utiliza grafos para gerar a função. O algoritmo é o que mais se aproxima do

[1]Um grafo randômico é caracterizado pelo fato de que as arestas são escolhidas dentre $\binom{|V|}{2}$ arestas igualmente prováveis, sendo $|V|$ o número de vértices do grafo.

[2]Um hipergrafo ou r-grafo é um grafo no qual as arestas conectam r vértices.

limite inferior de espaço para armazenar uma função *hash* perfeita mínima de ordem não preservada. Devido a este motivo ele também foi escolhido para ser avaliado e é discutido em mais detalhes na Seção 3.4, sendo referenciado pelo nome FCH.

Tanto os algoritmos da família MWHC-r [22, 7] quanto o algoritmo FCH [14] utilizam funções *hash* não perfeitas para gerar as funções *hash* perfeitas mínimas. Jenkins [20] apresenta um *survey* das principais funções *hash* não perfeitas disponíveis na literatura e propõe uma nova função *hash* mais rápida para ser computada.

1.3 Objetivos

O principal objetivo desta dissertação é realizar um estudo das principais funções *hash* perfeitas mínimas encontradas na literatura. São objetivos específicos os seguintes:

1. Identificar e comparar os melhores algoritmos conhecidos na literatura para gerar funções *hash* perfeitas mínimas. O estudo comparativo será realizado considerando os seguintes critérios:

   (a) tempo para gerar uma função *hash* perfeita mínima;

   (b) espaço necessário para armazenar uma função *hash* perfeita mínima;

   (c) tempo necessário para computar o endereço na tabela *hash* onde se encontra o registro referente a uma dada chave x.

2. Comparar a utilização de funções *hash* perfeitas mínimas para endereçar uma tabela *hash* com o método de *hashing* endereçamento aberto, o qual resolve colisões por *hashing* linear [21, 30]. As métricas utilizadas na comparação são:

   (a) tempo para computar o endereço de armazenamento ou recuperação de uma chave na tabela;

   (b) quantidade de memória interna consumida pelo sistema de busca para armazenar a tabela *hash* e a função que a indexa.

3. Identificar na literatura funções *hash* não perfeitas (funções que permitem colisões) que sejam eficientes. A intenção é melhorar a eficiência das funções *hash* perfeitas mínimas no que diz respeito ao tempo para computar o endereço de uma certa chave x na tabela *hash*. Isso porque, as funções *hash* perfeitas mínimas consideradas nesta dissertação são funções compostas por outras funções *hash* que não são perfeitas.

1.4 Contribuições

A principal contribuição desta dissertação é a realização de um estudo comparativo dos principais algoritmos disponíveis na literatura para gerar funções *hash* perfeitas mínimas. Podemos citar como contribuições específicas as seguintes:

1. Identificação, descrição e implementação dos principais algoritmos para gerar funções *hash* perfeitas mínimas (vide Capítulo 3). Os algoritmos mais eficientes, considerando o tempo de geração, são os da família de algoritmos MWHC-r apresentada na Seção 3.3. A função gerada é de ordem preservada e necessita de $O(n \log n)$ *bits* para ser armazenada. Já o algoritmo FCH, apresentado Seção 3.3, gera uma função de ordem não preservada e que é representável em $O(n)$ *bits*.

2. Identificação das vantagens e desvantagens de se utilizar funções *hash* perfeitas mínimas para endereçar uma tabela *hash* em comparação com o método endereçamento aberto que trata colisões por *hashing* linear. Na Seção 4.3 discutimos estes aspectos. Um resultado interessante é o fato de que com o método endereçamento aberto, o mecanismo de busca à um dicionário só será mais rápido para taxas de ocupação da tabela *hash* abaixo de 40% (inclusive). A taxa de ocupação ou fator de carga é definida pela razão entre o número de registros na tabela e o seu tamanho.

3. Identificação, descrição e implementação das principais funções *hash* não perfeitas disponíveis na literatura (vide Capítulo 2). Análise do impacto de cada uma das funções no tempo para se computar o endereço na tabela de uma chave x, tanto para funções *hash* perfeitas mínimas quanto para *hashing* linear. Determinação da função *hash* mais adequada para ser utilizada em cada um dos métodos.

1.5 Conceitos Básicos e Notação

Nesta Seção são apresentados a notação e os conceitos básicos utilizados na dissertação, os quais formalizam a noção intuitiva apresentada na Seção 1.1.

Definição 1 Seja $U = \{0, 1, \ldots, u-1\}$ um universo para algum inteiro positivo u escolhido de forma arbitrária e seja $S = \{s_1, s_2, \ldots, s_n\}$ um conjunto de n chaves distintas pertencentes a U. Cada chave é formada por k símbolos de um alfabeto Σ, sendo k uma constante. Uma chave identifica de forma única um registro de dados e o tamanho máximo de uma chave em S é denotado por t_{max}.

Definição 2 Uma função *hash* é uma função $h : U \to M$ que mapeia as chaves pertencentes a S para um dado intervalo de inteiros $M = [0..m-1]$, sendo $m \geq n$ o tamanho da tabela *hash*. Dada uma chave $s \in S$, a função *hash* computa o endereço $e \in M$ de armazenamento ou de recuperação na tabela *hash* do registro identificado por s.

Definição 3 Para quaisquer duas chaves distintas s_1 e $s_2 \in S$, uma colisão ocorre quando $h(s_1) = h(s_2)$.

Definição 4 Uma função *hash* universal é uma função $h : U \to M$ escolhida randomicamente de uma família de funções H, onde a probabilidade de ocorrência de colisão é $1/m$, sendo $m = |M|$.

Definição 5 Existem várias propostas para se tratar colisões em um método de *hashing*. A mais simples é a denominada de *hashing* linear. Nesta, a posição h_j de uma chave $s \in S$ na tabela é dada por:

$$h_j = (h(s) + j) \bmod m, \text{ para } 0 \leq j \leq m-1. \tag{1.1}$$

Definição 6 Uma função *hash* perfeita (FHP) é uma função injetora $h : U \to M$, o que significa que para todo s_1, $s_2 \in S$ tal que $s_1 \neq s_2$ então $h(s_1) \neq h(s_2)$, sendo $m \geq n$. Por ser injetora, h transforma cada chave de S em um único endereço na tabela *hash*, como mostrado na Figura 1.1 (a). Já que nenhuma colisão ocorre, cada registro pode ser recuperado da tabela com uma única tentativa.

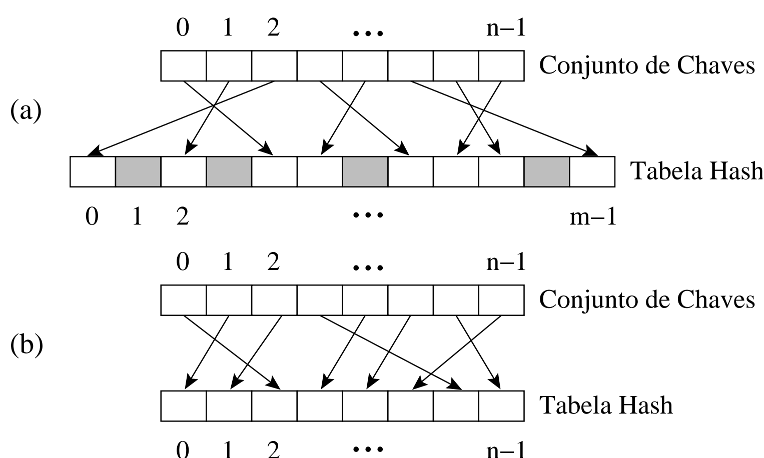

Figura 1.1: (a) Função *hash* perfeita. (b) Função *hash* perfeita mínima.

Definição 7 Uma função *hash* perfeita mínima (FHPM) é uma bijeção $h : U \to M$. Logo, pelo fato de h ser injetora, cada chave $s \in S$ é mapeada em um único endereço $e \in M$. Além disso, pelo fato de h ser sobrejetora, tem-se que $m = n$, como mostrado na Figura 1.1 (b).

Definição 8 Uma função *hash* perfeita h é de ordem preservada (FHPOP) se para qualquer par de chaves s_i e $s_j \in S$ então $h(s_i) < h(s_j)$ se e somente se $i < j$.

Definição 9 Fator de carga de uma tabela *hash* é definido pela razão $\alpha = n/m$. Quanto menor o valor de α maior a quantidade de espaço desperdiçado por posições não ocupadas na tabela *hash*. No entanto, a probabilidade de ocorrer colisões é menor.

Definição 10 Seja $G = (V, A)$, $|A| = n$, $|V| = cn$ um grafo randômico não direcionado sem *self-loops* e arestas múltiplas. O grafo G é gerado utilizando uma variação do **modelo uniforme** cunhado por Knuth [11]. Neste modelo, em cada passo um par não ordenado $a = \{u, v\}$ é gerado a partir de $\binom{|V|}{2}$ pares igualmente prováveis, onde u e $v \in V$. Se a aresta não direcionada a não é nem um *self-loop* e nem uma aresta múltipla, então ela é adicionada para G. Um *self-loop* acontece quando $u = v$. Para uma aresta $a_1 = \{u', v'\}$ já adicionada para G, uma aresta múltipla ocorre quando uma das condições ($u = u'$ e $v = v'$) ou ($u = v'$ e $v = u'$) é verdadeira.

A definição 10 é importante para a família de algoritmos MWHC-r [22], a qual é discutida em maiores detalhes na Seção 3.3. Uma outra definição importante é a de um hipergrafo, a qual vem a seguir.

Definição 11 Um hipergrafo ou r-grafo $G_r = (V, A)$ é definido por um conjunto de vértices V e um conjunto de arestas $A = \{a : a = \{v_1, v_2, \ldots v_r\}\}$, sendo que $v_1, v_2, \ldots v_r \in V$. Em outras palavras, cada aresta em A conecta r vértices de V.

Uma outra definição importante para a família de algoritmos MWHC-r é a de um r-grafo acíclico, a qual é apresentada abaixo.

Definição 12 Um hipergrafo ou r-grafo $G_r = (V, A)$ é acíclico se, e somente se, obtivermos um grafo vazio a partir de uma seqüência de remoções das arestas de G_r que contenham vértices de grau 1.

1.6 Estrutura da Dissertação

Esta dissertação está estruturada da seguinte forma: No Capítulo 2 apresentamos as principais funções *hash* não perfeitas encontradas na literatura. A escolha das funções foi baseada na eficiência para computar o endereço na tabela e na uniformidade das funções. No Capítulo 3 descrevemos os principais algoritmos da literatura para gerar FHPMs. No Capítulo 4 apresentamos o estudo comparativo realizado. Finalmente, as conclusões e os trabalhos futuros são apresentados no Capítulo 5.

Capítulo 2

Funções *Hash*

Neste capítulo descrevemos as funções *hash* não perfeitas utilizadas na geração de FHPMs e no método de *hashing* que armazena o dicionário por endereçamento aberto e trata colisões por *hashing* linear.

Segundo Knuth [21], uma boa função *hash* é aquela que: (i) é simples de ser computada e (ii) minimiza o número de colisões, isto é, para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer. Se as chaves fossem verdadeiramente randômicas, bastaria extrair alguns *bits* delas e usá-los para compor o valor da função *hash*. Mas na prática sempre é necessário que o valor da função *hash* seja dependente de todos os *bits* da chave para satisfazer a propriedade (ii). Com base nestes dois critérios, selecionamos três funções *hash* para serem avaliadas neste trabalho:

1. Função *hash* universal apresentada em [30].

2. Função *hash* proposta por Zobrist [31].

3. Função *hash* proposta por Jenkins [20].

As duas primeiras foram selecionas por serem simples de computar e por espalharem uniformemente o conjunto de chaves nas entradas da tabela. A terceira foi escolhida por ser computada mais eficientemente que as anteriores, apesar do espalhamento das chaves não ser tão uniforme.

2.1 Função *Hash* Universal

Nesta seção apresentamos uma função *hash* universal hu escolhida randomicamente de uma família de funções HU, onde a probabilidade de ocorrência de colisão é $1/m$, sendo

m o número de entradas na tabela *hash*. Considerando o conjunto S da Definição 1 da Seção 1.5 e um conjunto de pesos $P = \{P_i \,/\, 0 \leq i < t_{max}\}$, onde cada P_i é um número inteiro escolhido aleatoriamente dentro do intervalo $M = [0, m-1]$, a função $hu : U \rightarrow M$ é definida por:

$$hu(s \in S) = \left(\sum_{i=0}^{t_{max}-1} P_i \times s_i \right) \bmod m.$$

Cada conjunto distinto de pesos caracteriza uma função $hu \in HU$. A Figura 2.1 apresenta o programa utilizado para gerar os pesos de uma certa função hu.

```
typedef unsigned long TipoInt;

void GenPesosHashUniversal (TipoInt * P, TipoInt t_max , TipoInt m)
{
  register TipoInt i;
  for (i = 0; i < t_max; i++)
    P[i] = (TipoInt) (m*(rand()/(RAND_MAX+1.0)));
}
```

Figura 2.1: Geração do conjunto de pesos P da função *hash* universal.

Agora mostraremos que a família de funções HU é realmente uma família universal. Sejam $x = <x_0, x_1, \ldots, x_{t_{max}}>$ e $y = <y_0, y_1, \ldots, y_{t_{max}}>$ duas chaves distintas de S. Considerando um dado conjunto de pesos $P = \{P_i \,/\, 0 \leq i < t_{max}\}$, o tamanho da família HU é $m^{t_{max}}$, pois para cada P_i existem m valores possíveis. Considerando que $x_0 \neq y_0$, vamos contar o número de funções em HU nas quais $hu(x) = hu(y)$. Assim, temos que:

$$\begin{aligned} \sum_{i=0}^{t_{max}-1} P_i \times x_i &\equiv \sum_{i=0}^{t_{max}-1} P_i \times y_i \pmod{m} \\ P_0 \times (x_0 - y_0) &\equiv -\sum_{i=1}^{t_{max}-1} P_i \times (x_i - y_i) \pmod{m} \end{aligned} \tag{2.1}$$

Agora vamos isolar P_0 na Eq. (2.1). Para isto, temos que fazer

$$(x_0 - y_0) \times (x_0 - y_0)^{-1} \equiv 1 \pmod{m}.$$

Para existir o inverso $\pmod{m}$, o maior divisor comum entre $(x_0 - y_0)$ e m deve ser um, obrigatoriamente. Uma forma de forçar esta restrição é considerar m um número primo.

Assim,

$$P_0 \equiv -\sum_{i=1}^{t_{max}-1} P_i \times (x_i - y_i)(x_0 - y_0)^{-1} \pmod m \tag{2.2}$$

Para cada P_i na Eq. (2.2), sendo $1 \leq i \leq t_{max} - 1$, m valores distintos são possíveis e para cada um deles podemos determinar o valor de P_0. Dessa forma, existem $m^{t_{max}-1}$ funções nas quais $hu(x) = hu(y)$. Conseqüentemente, a probabilidade de ocorrência de colisões para as funções da família HU é:

$$\frac{m^{t_{max}-1}}{m^{t_{max}}} = \frac{1}{m}$$

Logo, HU é uma família universal.

A função *hash* universal é muito simples de ser implementada, como ilustra a Figura 2.2. Segundo Knuth [21, p. 516], para melhorar a uniformidade da função hu, o número primo escolhido para m não deve possuir o formato $b^i \pm j$, onde b é a base do conjunto de caracteres (geralmente $b = 64$ para BCD, 128 para ASCII, 256 para EBCDIC, entre outros), i e j são pequenos inteiros.

```
TipoInt hu (const char * chave, TipoInt * P, TipoInt m)
{  register TipoInt i;
   register unsigned long long soma  = ((unsigned char)chave[0]) * P[0];
   register TipoInt tamanho = strlen (chave);
   for (i = 1; i < tamanho; i++)
     soma += ((unsigned char)chave[i]) * P[i];
   return ((TipoInt)(soma % m));
}
```

Figura 2.2: Implementação da função *hash* universal.

2.2 Função *Hash* de Zobrist

Nesta seção apresentamos a função *hash* hz proposta por Zobrist [31]. Para computar hz é necessário o mesmo número de adições da função *hash* universal, mas nenhuma multiplicação é efetuada. Isto faz com que hz seja computada mais eficientemente do que a função *hash* universal.

Considerando o conjunto S da Definição 1 da Seção 1.5 e um conjunto de pesos $P = \{P_{i,j} \ / \ 0 \leq i < t_{max} \text{ e } 0 \leq j < |\Sigma|\}$, sendo cada $P_{i,j}$ um número inteiro escolhido aleatoriamente dentro do intervalo $M = [0, m-1]$, a função $hz : U \rightarrow M$ é definida por:

$$hz(s \in S) = \left(\sum_{i=0}^{t_{max}-1} P_{i,s_i} \right) \bmod m.$$

Cada conjunto distinto de pesos caracteriza uma função hz distinta. A Figura 2.3 apresenta o programa utilizado para gerar os pesos de uma certa função hz.

```
void GenPesosHashZobrist (TipoInt ** P, TipoInt t_max, TipoInt TamAlfabeto,
                          TipoInt m)
{ TipoInt i, j, t;
  for (i = 0; i < t_max; i++)
    for (j = 0; j < TamAlfabeto; j++)
      P[i][j] = (TipoInt) (m*(rand()/(RAND_MAX+1.0)));
}
```

Figura 2.3: Geração do conjunto de pesos P da função hz.

A probabilidade de ocorrência de colisão na função hz também é $1/m$. Para mostrar isto, basta provar que hz também é uma função *hash* universal. Sejam HZ a família de funções da qual hz é escolhida randomicamente, $x = < x_0, x_1, \ldots, x_{t_{max}} >$ e $y = < y_0, y_1, \ldots, y_{t_{max}} >$ duas chaves distintas de S. Considerando um dado conjunto de pesos $P = \{P_{i,j} \ / \ 0 \leq i < t_{max} \text{ e } 0 \leq j < |\Sigma|\}$, o tamanho da família HZ é $m^{t_{max} \times |\Sigma|}$, pois para cada símbolo do alfabeto Σ que pode ocupar a $i-$ésima posição de uma chave existem m valores possíveis de serem gerados para $P_{i,j}$. Considerando que $x_0 \neq y_0$, vamos contar o número de funções em HZ nas quais $h(x) = h(y)$. Assim, temos que:

$$\begin{aligned} \sum_{i=0}^{t_{max}-1} P_{i,x_i} &\equiv \sum_{i=0}^{t_{max}-1} P_{i,y_i} \pmod{m} \\ (P_{0,x_0} - P_{0,y_0}) &\equiv -\sum_{i=1}^{t_{max}-1} (P_{i,x_i} - P_{i,y_i}) \pmod{m} \end{aligned} \tag{2.3}$$

Isolando P_{0,x_0} na Eq. (2.3) temos:

$$P_{0,x_0} \equiv P_{0,y_0} - \sum_{i=1}^{t_{max}-1} (P_{i,x_i} - P_{i,y_i}) \pmod{m} \tag{2.4}$$

Assim, sempre é possível encontrar um valor para P_{0,x_0} que satisfaça a Eq. (2.4), independente do conjunto de pesos utilizados para os $P_{i,j}$, sendo $j \neq x_0$. Como somente um índice de P é fixado e existem m valores possíveis para as demais $t_{max} \times |\Sigma| - 1$ posições, então, existem $m^{t_{max} \times |\Sigma| - 1}$ funções nas quais $h(x) = h(y)$. Conseqüentemente, a probabilidade de ocorrência de colisões para as funções da família HZ é:

$$\frac{m^{t_{max} \times |\Sigma| - 1}}{m^{t_{max} \times |\Sigma|}} = \frac{1}{m}$$

Logo, HZ é uma família universal.

A função *hash* de Zobrist [31] é ainda mais simples de ser implementada, como ilustra a Figura 2.4. Embora nenhuma multiplicação seja efetuada, a quantidade de memória interna para armazenar hz é $O(t_{max} \times |\Sigma|)$ enquanto que para a função *hash* universal é $O(t_{max})$.

```
TipoInt hz (const char *chave, TipoInt ** P, TipoInt m)
{  register TipoInt i;
   register unsigned long long soma = P[0][(unsigned char)chave[0]];
   register TipoInt tamanho = strlen (chave);
   for (i = 1; i < tamanho; i++)
     soma += P[i][(unsigned char)chave[i]];
   return ((TipoInt) soma % m);
}
```

Figura 2.4: Implementação da função hz.

2.3 Função *Hash* de Jenkins

Nesta seção apresentamos a função *hash* hj proposta por Jenkins [20]. A função foi projetada para atender os seguintes requisitos:

1. Permitir que as chaves sejam arranjos de caracteres de tamanho variável. O tamanho dos arranjos deve variar de 8 a 200 *bytes*.

2. Ser mais eficiente que as funções existentes.

3. Permitir qualquer tamanho de tabela, inclusive potência de 2.

4. Distribuir uniformemente as chaves nas entradas da tabela.

Para satisfazer o requisito 4 foi criada uma certa dependência entre as chaves do conjunto S. A função recebe um valor randômico que serve de semente inicial para se computar o valor referente à primeira chave de S. Para computar o valor referente a segunda chave, o resultado da aplicação de hj sobre a primeira chave é utilizado como semente, e assim sucessivamente. Esta dependência inviabiliza a aplicação de hj no contexto desta dissertação. Para evitar este problema, nós utilizamos a mesma semente inicial para todas as chaves. Uma conseqüência direta disto é que apenas um número inteiro de 4 *bytes* é necessário para representar hj, embora o número de colisões ocasionadas por ela seja maior que o número de colisões ocasionadas pelas funções descritas nas seções 2.1 e 2.2. No capítulo 4 apresentamos os experimentos realizados para discutir o impacto da aplicação da função hj nos algoritmos considerados.

A Figura 2.5 apresenta um modelo que se aplica a diversas funções *hash* da literatura, inclusive à função hj. Primeiramente, um estado interno é estabelecido e inicializado. O estado interno é representado por três variáveis inteiras de 4 *bytes* (a, b e c). Em seguida, blocos de texto da chave contendo 12 *bytes* são combinados com o estado interno através de adições. Após isto, um embaralhamento dos *bits* do estado interno é realizado com o objetivo de mapear chaves ligeiramente distintas em endereços distintos na tabela *hash*. A Figura 2.6 apresenta o código para realizar o embaralhamento. Por fim, um pós processamento é efetuado e um inteiro de 4 *bytes* é extraído do estado interno para representar o valor da função.

```
Inicializa (EstadoInterno);
for each BlocoDaChave do
  Combinar (EstadoInterno, BlocoDaChave);
  Embaralhar (EstadoInterno);
return PosProcessamento (EstadoInterno);
```

Figura 2.5: Modelagem da função hj.

Agora mostraremos como o estado interno é combinado com os blocos de texto da chave. Suponha que o estado interno é inicializado da seguinte forma:

a				b				c			
158	055	121	185	158	055	121	185	113	177	053	037

Tabela 2.1: Inicialização do estado interno.

```
#define Embaralhar (a,b,c) \
{ \
  a -= b; a -= c; a ^= (c >> 13); \
  b -= c; b -= a; b ^= (a << 8); \
  c -= a; c -= b; c ^= (b >> 13); \
  a -= b; a -= c; a ^= (c >> 12);  \
  b -= c; b -= a; b ^= (a << 16); \
  c -= a; c -= b; c ^= (b >> 5); \
  a -= b; a -= c; a ^= (c >> 3);  \
  b -= c; b -= a; b ^= (a << 10); \
  c -= a; c -= b; c ^= (b >> 15); \
}
```

Figura 2.6: Implementação do embaralhamento dos *bits* do estado interno.

onde cada bloco de três dígitos das variáveis a, b e c representa o valor em decimal armazenado em cada um de seus *bytes*. Nas tabelas apresentadas nesta seção, os *bytes* mais à esquerda são os mais significativos. O valor inicialmente atribuído para a e b foi determinado experimentalmente por Jenkins [20] visando impedir o mapeamento de chaves distintas com muitos *bits* iguais a zero no mesmo endereço da tabela *hash*. Devido a este fato, o valor foi denominado Razão de Ouro e corresponde ao valor 0x9E3779B9 em hexadecimal. Já o valor inicial de c corresponde à semente randômica da função. É esta semente que diferencia uma função de outra.

Suponha que o seguinte bloco de texto B será combinado com o estado interno:

Bloco de texto com 12 *bytes* e seus respectivos códigos ASCII											
F	A	B	I	A	N	O	B	O	T	E	L
070	065	066	073	065	078	079	066	079	084	069	076

Tabela 2.2: Bloco de texto a ser combinado com o estado interno.

A Tabela 2.3 ilustra como o bloco de texto B da Tabela 2.2 é combinado com o estado interno da Tabela 2.1. A combinação é determinada pelas seguintes equações:

$$\mathrm{a} = \mathrm{a} + B[0] + (B[1] << 8) + (B[2] << 16) + (B[3] << 24) \quad (2.5)$$

$$\mathrm{b} = \mathrm{b} + B[4] + (B[5] << 8) + (B[6] << 16) + (B[7] << 24) \quad (2.6)$$

$$\mathrm{c} = \mathrm{c} + B[8] + (B[9] << 8) + (B[10] << 16) + (B[11] << 24) \quad (2.7)$$

A operação $B[i]$, com $0 \leq i \leq 11$, converte o caractere na posição i de B para um número inteiro de 4 *bytes* que representa o valor do seu código ASCII. A operação $<<$ move os

bits para a esquerda e entra com zeros à direita (por exemplo, $B[1] << 8$ move o valor armazenado em $B[1]$ 8 *bits* para a esquerda e entra com 8 zeros à direita). O resultado de cada equação é um número inteiro de 4 *bytes*. As 5 primeiras linhas da Tabela 2.3 correspondem as parcelas da soma e a última linha o resultado da soma das parcelas.

a					b					c			
158	055	121	185		158	055	121	185		113	177	053	037
000	000	000	**070**		000	000	000	**065**		000	000	000	**079**
000	000	**065**	000	+	000	000	**078**	000	+	000	000	**084**	000
000	**066**	000	000		000	**079**	000	000		000	**069**	000	000
073	000	000	000		**066**	000	000	000		**076**	000	000	000
231	121	186	255		224	134	199	250		189	246	137	116

Tabela 2.3: Combinação do bloco de texto com o estado interno através de adições.

A Figura 2.7 apresenta a implementação da função hj. Um problema que surge durante a implementação é quando os comprimentos das chaves não são múltiplos de 12. Para resolver este problema, basta utilizar o comando **switch** da linguagem C para indicar em qual parcela das Eq. (2.5), (2.6) e (2.7) a combinação do novo estado interno com o último bloco de texto deve iniciar. Para não mapear chaves que se diferem somente no comprimento (por exemplo, "aaaa" e "aaaaa") no mesmo endereço da tabela *hash*, o *byte* menos significativo de c é adicionado ao tamanho da chave ao invés de ser adicionado ao valor armazenado em $B[8]$, como ocorre no laço **while**. Assim, a Eq. (2.7) é alterada para: $c = c + \text{tamanho} + (B[8] << 8) + (B[9] << 16) + (B[10] << 24)$, sendo que as três últimas parcelas podem ou não ser adicionadas a c, dependendo do tamanho do último bloco processado.

O tamanho da tabela *hash* pode ser qualquer valor. No entanto, hj é ainda mais eficiente para tamanhos que são potências de 2. Por exemplo, suponha que a tabela tenha capacidade para 1024 registros. Como o valor de hj é um inteiro de 4 *bytes* e para endereçar 1024 registros precisamos de apenas 10 *bits*, então, basta realizar uma operação de *and bit a bit* (&) do valor retornado por hj com a máscara 1111111111, a qual é dada por hashmask (10). Porém, caso o tamanho não seja uma potência de 2, por exemplo $m = 1021$, após realizar a operação de *and bit a bit* com a máscara, se o resultado for maior ou igual a m uma operação de ou-exclusivo (^) com o tamanho da tabela *hash* deve ser efetuada para limitar o resultado de hj no intervalo $[0, m-1]$.

```
#define hashsize (nbits) ((TipoInt)1<<(nbits))
#define hashmask (nbits) (hashsize (nbits)-1)

TipoInt hj (const char * B, TipoInt sementeInicial, TipoInt m) {
  register TipoInt a, b, c, blocoAtual, tamanho;
  register TipoInt nbits; /* Número de bits para armazenar m */
  /* Inicialização do estado interno, representado pelas variáveis a, b e c */
  a = b = 0x9e3779b9; /* A razão de ouro, um valor arbitrário      */
  c = sementeInicial; /* Valor randômico inicial para a função hj */
  nbits = ceil(log(m)/log(2));
  tamanho = strlen (B);
  blocoAtual = tamanho;
  while (blocoAtual >= 12) {
    /* Combinando o estado interno através de adições */
    a += (B [0] + ((TipoInt)B [1] << 8) + ((TipoInt)B [2] << 16) +
          ((TipoInt)B [3] << 24));
    b += (B [4] + ((TipoInt)B [5] << 8) + ((TipoInt)B [6] << 16) +
          ((TipoInt)B [7] << 24));
    c += (B [8] + ((TipoInt)B [9] << 8) + ((TipoInt)B [10] << 16) +
          ((TipoInt)B [11] << 24));
    Embaralhar (a,b,c);         /* Embaralhamento do estado interno */
    B += 12; blocoAtual -= 12; /* Próximo bloco a ser manipulado   */
  }

  /* Manipulação dos últimos 11 bytes */
  c  +=tamanho; /* O primeiro byte de c é reservado para o tamanho     */
  switch (blocoAtual) {
    case 11: c += ((TipoInt)B [10] << 24);
    case 10: c += ((TipoInt)B [09] << 16);
    case 9 : c += ((TipoInt)B [08] << 8);
    case 8 : b += ((TipoInt)B [07] << 24);
    case 7 : b += ((TipoInt)B [06] << 16);
    case 6 : b += ((TipoInt)B [05] << 8);
    case 5 : b +=B [04];
    case 4 : a += ((TipoInt)B [03] << 24);
    case 3 : a += ((TipoInt)B [02] << 16);
    case 2 : a += ((TipoInt)B [01] << 8);
    case 1 : a +=B [0];
    /* case 0: Não há nada para adicionar */
   }
   Embaralhar (a,b,c);
   c = (c & hashmask (nbits));
   return ((c >=m) ? c ^ m: c);
}
```

Figura 2.7: Implementação da função hj.

As características da função hj que a torna mais eficiente que as demais funções são:

1. O embaralhamento dos *bits* é efetuado sobre três registradores de 4 *bytes*, ao invés de ser realizado com registradores de apenas 1 *byte*.

2. A etapa de combinação é realizada sobre blocos de texto de 12 *bytes* o que reduz o *overhead* do laço **while**.

3. O comando **switch** combina o bloco de texto final de tamanho variável com o estado interno sem a necessidade de introduzir o *overhead* de um laço.

2.4 Armazenamento das Funções em Memória Externa

O armazenamento em memória externa das três funções *hash* apresentadas neste capítulo pode ser padronizado utilizando um gerador de números pseudo-aleatórios independente de arquitetura, pois as três funções dependem de um gerador de números pseudo-aleatórios. Os geradores de números pseudo-aleatórios necessitam de um número inteiro inicial que serve de semente para a geração dos demais. Portanto, basta armazenar a semente fornecida para o gerador de números pseudo-aleatórios que a mesma seqüência de números utilizados nas funções pode ser reproduzida em memória interna.

A Figura 2.8 apresenta o trecho de código utilizado para gerar a semente e atribuí-la ao gerador de números pseudo-aleatórios.

```
#include <stdlib.h>
#include <sys/time.h>
...
struct timeval tempo;
TipoInt semente;
...
gettimeofday (&tempo, NULL);
semente = (TipoInt)(tempo.tv_sec + 1001*tempo.tv_usec);
srand (semente);
...
```

Figura 2.8: Geração da semente para o gerador de números pseudo-aleatórios.

Capítulo 3

Algoritmos Avaliados

Neste capítulo apresentamos a estrutura de dados que representa a tabela *hash* e os algoritmos avaliados. Após apresentarmos a estrutura de dados, descrevemos o método de *hashing* endereçamento aberto com tratamento de colisão através de *hashing* linear. Em seguida, descrevemos os algoritmos que constituem o estado da arte na geração de FHPMs. A família de algoritmos MWHC-r apresentada na seção 3.3 possui os algoritmos mais rápidos no que diz respeito ao tempo de geração de uma FHPM. Já o algoritmo FCH apresentado na seção 3.4 é o melhor algoritmo da literatura no que diz respeito ao espaço para armazenar as FHPMs geradas.

3.1 Estrutura da Tabela *Hash*

A Figura 3.1 ilustra a estrutura de dados utilizada na representação da tabela *hash*. A estrutura de dados é composta de duas partes:

1. Conjunto de Chaves S: é um pedaço contíguo de memória que contém todas as chaves do dicionário. As chaves são separadas por caracteres nulos (\0);

2. Tabela *Hash*: é um arranjo com um número fixo m de registros. Cada registro contém um apontador para o início da chave que o identifica no Conjunto de Chaves.

Existem outras formas de representar um dicionário, como apresentado em [29]. Nós optamos pela estrutura apresentada na Figura 3.1 pelo fato de sua implementação ser simples e, principalmente, por reduzir o *overhead* de espaço no armazenamento do dicionário.

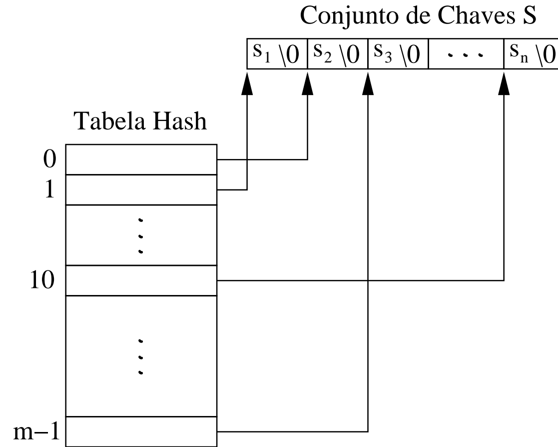

Figura 3.1: Estrutura de dados que representa a tabela *hash*.

3.2 Endereçamento Aberto com *Hashing* Linear

No método endereçamento aberto os registros são armazenados em uma tabela de tamanho $m > n$ e as colisões são resolvidas utilizando as próprias posições vazias da tabela. O método envolve dois passos: (i) utilização de uma função *hash* para mapear as chaves de S para números inteiros; (ii) resolução de colisões.

No Capítulo 4 discutimos o impacto de cada uma das funções *hash* apresentadas no Capítulo 2 no passo (i) do método endereçamento aberto. Agora vamos ilustrar como se dá a resolução de colisões. Uma forma simples de resolver colisões e que tem um bom desempenho na prática é através do *hashing* linear. Como apresentado na Definição 5 da Seção 1.5, a posição h_j de uma chave $s \in S$ na tabela é dada por:

$$h_j = (h(s) + j) \bmod m, \text{ para } 0 \leq j \leq m - 1.$$

Considerando um conjunto de chaves $S = \{F, A, B, I, N, O\}$ constituído de um único caractere e que a i-ésima letra do alfabeto é representada pelo número i, vamos mostrar como funciona o *hashing* linear para uma tabela *hash* T de tamanho $m = 7$ e uma função *hash* $h(s \in S) = s \mod m$. Por exemplo, ao aplicarmos h a cada chave de S obtemos: $h(F) = h(6) = 6$, $h(A) = h(1) = 1$, $h(B) = h(2) = 2$, $h(I) = h(9) = 2$, $h(N) = h(14) = 0$, $h(O) = h(15) = 3$. O resultado da inserção do conjunto S em T é apresentado na Figura 3.2. A chave I colide com a chave B, logo sua posição é dada por $(h(I) + 1) \bmod 7 = 3$. A colisão de I com B faz com que O colida com I, assim a chave O passa a ocupar a posição $(h(I) + 1) \mod 7 = 4$.

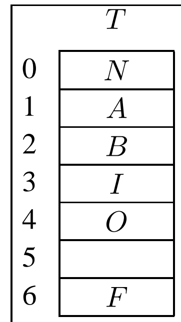

Figura 3.2: Endereçamento aberto

A eficiência para se pesquisar uma certa chave $s \in S$ no método endereçamento aberto com resolução de colisões através de *hashing* linear é dependente do número de colisões ocorridas durante a busca. Quanto mais cheia estiver a tabela *hash* maior a probabilidade de ocorrência de colisões. Segundo Knuth [21], o número de comparações $C(n)$ necessárias para localizar s é aproximadamente:

- Pesquisa com sucesso: $C(n) = \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)\right) \cdot$
- Pesquisa sem sucesso: $C(n) = \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) \cdot$

Expandindo as equações acima através de série de Taylor para um fator de carga $\alpha \simeq 0$, podemos obter uma aproximação para o número de comparações necessárias para encontrar uma chave quando o fator de carga é pequeno (próximo de zero). Assim,

- Pesquisa com sucesso: $C(n) = 1 + \frac{\alpha}{2} = 1 + \frac{n}{2m} \cdot$
- Pesquisa sem sucesso: $C(n) = 1 + \alpha = 1 + \frac{n}{m} \cdot$

O principal problema com o método descrito é um pior caso de pesquisa com complexidade $O(n)$, o qual pode ocorrer se a função não conseguir espalhar os registros uniformemente nas entradas da tabela. Isto pode ocasionar a formação de uma longa lista linear, deteriorando o tempo médio de pesquisa.

3.3 A Família de Algoritmos MWHC-r

Nesta seção descrevemos a família de algoritmos MWHC-r proposta por Majewski, Wormald, Havas e Czech [22], na qual estão inseridos algoritmos práticos e eficientes para gerar FHPMs de ordem preservada a partir de r-grafos randômicos. Os algoritmos são ótimos

tanto no tempo de geração das FHPMs quanto no espaço necessário para armazenar as funções.

O objetivo dos membros da família MWHC-r é resolver o seguinte **problema de atribuição perfeita**: Para um dado r-grafo $G_r = (V, A)$, onde $|A| = n$ e cada aresta $a \in A$ é um subconjunto de tamanho r do conjunto de vértices V, o problema consiste em encontrar uma função $g : V \to \{0, 1, \ldots, n-1\}$ de forma que a função $h : E \to \{0, 1, \ldots, n-1\}$ definida como

$$h(a) = g(v_1) \oplus g(v_2) \oplus \cdots \oplus g(v_r) \tag{3.1}$$

seja uma bijeção, onde $a = \{v_1, v_2, \ldots, v_r\}$ é uma aresta do conjunto E e o símbolo $\oplus$ representa a operação de ou exclusivo. Isso significa que estamos procurando por uma atribuição de valores para os vértices em V de forma que, para cada aresta em A, o ou exclusivo dos valores associados a seus vértices deve ser um único inteiro no intervalo $[0, n-1]$.

Majewski et al. [22] mostraram que o problema de atribuição perfeita pode ser resolvido em tempo ótimo para um r-grafo acíclico através da utilização da abordagem MOS. Os algoritmos da família MWHC-r executam os passos de mapeamento e ordenação simultaneamente. Nestes dois passos, o r-grafo acíclico G_r é gerado a partir do conjunto S (vide Seção 3.3.1). Já no passo de pesquisa, os valores da função g são definidos a partir de G_r para gerar a FHPM mostrada na Eq. (3.1) (vide Seção 3.3.2).

3.3.1 Mapeamento e Ordenação

O passo de mapeamento recebe como entrada o conjunto de chaves S e gera um r-grafo $G_r = (V, E)$, sendo $|E| = |S| = n$ e $|V| = cn$. O grafo é gerado randomicamente de acordo com a Definição 10 da Seção 1.5 e deve possuir as seguintes propriedades: (i) não ser direcionado; (ii) não conter *self-loops* e arestas múltiplas e (iii) ser acíclico. Uma observação a ser feita é que existe uma correspondência entre as chaves do conjunto S e as arestas do conjunto E.

O grafo é gerado utilizando r-funções *hash* $h_1, h_2, \ldots, h_r$. Cada uma destas funções mapeia as chaves de S para inteiros no intervalo $[0, |V| - 1]$ e são funções não perfeitas como as descritas no Capítulo 2. Para cada chave $s \in S$, a aresta correspondente $a = \{h_1(s), h_2(s), \ldots, h_r(s)\}$ é adicionada para E. Um *self-loop* acontece quando $h_i(s) = h_j(s)$, sendo $i \neq j$. Para evitar *self-loops*, a função h_j é modificada através da adição modulo $|V|$ de um valor randômico no intervalo $[1, |V| - 1]$ (vide Seção 3.3.3 para detalhes). A

probabilidade de ocorrência das arestas múltiplas tende a zero quando n tende a infinito [6, Lema 3.3, página 582]. Assim, nenhum teste é feito para a verificação da propriedade (ii) acima. Além disso, caso o grafo contenha arestas múltiplas, isto será detectado durante a verificação da propriedade (iii).

Um algoritmo ótimo, $O(n)$, para a verificação da propriedade (iii) é apresentado em [17]. A Figura 3.3 ilustra o funcionamento do algoritmo, passo a passo, para um 2-grafo G_2 de 6 vértices e 5 arestas. Cada vértice v da Figura 3.3(a) deve ser visitado uma única vez. Os vértices são visitados na ordem ascendente de seus rótulos e a cor cinza serve para indicar os vértices já visitados. Se v é de grau 1, então, a aresta $a \in E$ que contém v deve ser removida de G_2. Cada aresta removida é adicionada a uma pilha P que será utilizada no passo de pesquisa (vide Seção 3.3.2). A Figura 3.3(b) ilustra o momento que o vértice $v = 0$ é visitado, a aresta $a = \{0, 2\}$ é removida de G_2 e inserida em P. Após a remoção da aresta a, caso algum outro vértice $u \in a$ se torne de grau 1, o processo é repetido recursivamente para u. Este fato é ilustrado na Figura 3.3(c) e (d), pois após o vértice $v = 1$ ser visitado e a aresta $a = \{1, 2\}$ ser inserida em P, o vértice $u = 2$ se torna de grau 1. Logo, o vértice 2 é visitado e a aresta $\{2, 3\}$ é inserida em P. O algoritmo termina quando todos os vértices são visitados. O grafo é acíclico se todas as arestas forem removidas do grafo e inseridas na pilha, como ilustra a Figura 3.3(f).

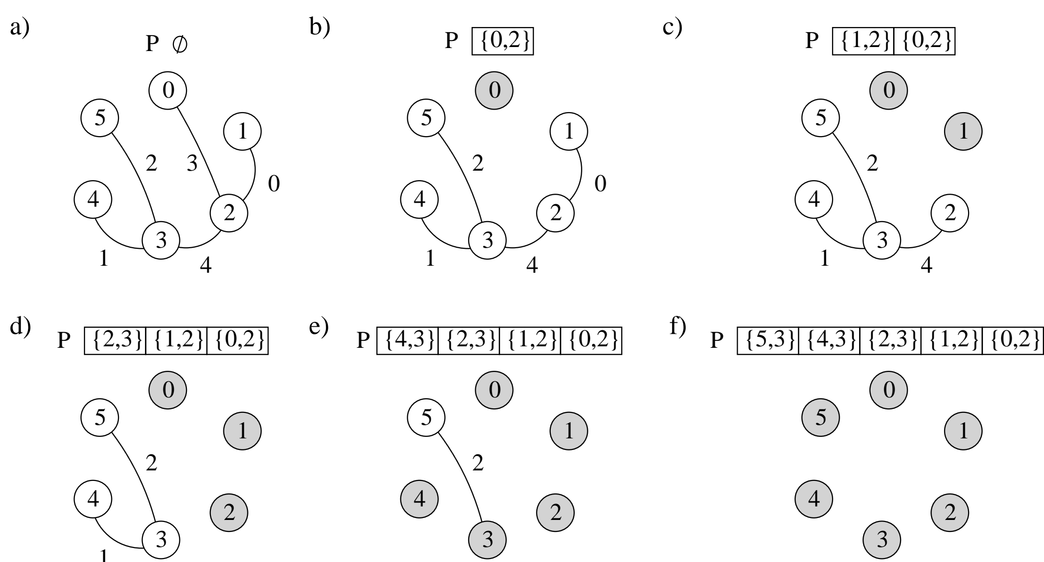

Figura 3.3: Teste de aciclicidade

A Figura 3.4 apresenta um pseudocódigo para o algoritmo de teste de aciclicidade de um r-grafo G_r. Majewski et al. [22, Teorema 2.2] mostram que a complexidade deste

algoritmo é $O(rn + |V|)$. Como $|V| = cn$, então a complexidade é $O(rn + cn)$ que é igual a $O(n)$, uma vez que c e r são constantes.

```
ExisteCiclo (Gr , P) {
  for (v = 0; v < |V|; v++)
    if (Grau (v) == 1)
      RemoverArestaRecursivamente (Gr , v , P);
  if (Vazio (E)) return "grafo acíclico";
  else return "grafo cíclico";
}

RemoverArestaRecursivamente (Gr , v , P){
  Remover de Gr a aresta a que contém o vértice v;
  Empilhar (a , P);
  for (u = 0; u < r; u++)
    if (Grau (a[u]) == 1)
      RemoverArestaRecursivamente (Gr , a[u] , P);
}
```

Figura 3.4: Pseudocódigo para o teste de aciclicidade de um r-grafo G_r.

Para garantir a aciclicidade do r-grafo, no passo de mapeamento o algoritmo repetidamente seleciona as funções $h_1, h_2, \ldots, h_r$ até o r-grafo correspondente G_r ser acíclico. Para tornar possível a geração de um r-grafo acíclico $G_r = (V, E)$, onde $|V| = c|E|$, temos que determinar o valor da constante c de forma que r-grafos acíclicos sejam dominantes no espaço de todos os r-grafos randômicos possíveis (o número de r-grafos possíveis com $|V|$ vértices e $|E|$ arestas é $\binom{\binom{|V|}{r}}{|E|}$).

Para $r = 2$, Majewski et al. [22, Teorema 4.2] mostram que a probabilidade p de um 2-grafo ser acíclico é

$$p = e^{1/c}\sqrt{\frac{c-2}{c}}, \tag{3.2}$$

para $c > 2$ e n tendendo a infinito. A probabilidade p é alta mesmo para valores pequenos de c. Por exemplo, para $c = 2.09$ a probabilidade é $p > 1/3$. Conseqüentemente, o número esperado de iterações do passo de mapeamento para obter um 2-grafo acíclico é menor que 3 (o número de iterações é $N_i(X) = 1/p$, onde X é uma variável randômica que conta o número de iterações para gerar um 2-grafo acíclico).

Para $r \geq 3$, Majewski et al. [22, Claim 4.3] mostram que o valor de $c = c_r$ (o valor de c se transforma em uma função de r) para obter um r-grafo acíclico com **alta probabilidade**[1] é $c_r = r/\gamma_r$, onde

$$\gamma_r = \min_{x>0} \left\{ \frac{x}{(1-e^{-x})^{r-1}} \right\}. \tag{3.3}$$

Os autores obtiveram $\gamma_3 \simeq 2.45541$, $\gamma_4 \simeq 3.08912$ e $\gamma_5 \simeq 3.50890$ para $r = 3, 4, 5$, respectivamente. Assim, os respectivos valores de c_3, c_4 e c_5 são 1.22179, 1.29487 e 1.42495. Nesta dissertação consideramos apenas os algoritmos onde $r = 2$ e $r = 3$, ou seja MWHC-2 e MWHC-3, pois são os mais eficientes da família MWHC-r em termos de espaço e de tempo para gerar uma FHPM.

Por fim, no passo de ordenação é associado a cada aresta a um único número $h(a) \in [0, n-1]$ na ordem das chaves em S. O objetivo é obter uma FHPM de ordem preservada. Os números associados às arestas correspondem aos endereços da tabela *hash*. O passo de ordenação é efetuado durante a construção do grafo. Considerando o exemplo da Figura 3.3(a), às arestas $\{1,2\}$, $\{3,4\}$, $\{3,5\}$, $\{0,2\}$ e $\{2,3\}$ foram associados os endereços 0, 1, 2, 3, 4 e 5, respectivamente.

A Figura 3.5 apresenta um pseudocódigo para os passos de mapeamento e ordenação. Escolhendo adequadamente o valor da constante c, o número de iterações para se gerar um grafo acíclico é $O(1)$. Uma vez que a complexidade de verificação de aciclicidade é $O(n)$, então, a complexidade dos passos de mapeamento e ordenação também é $O(n)$.

3.3.2 Pesquisa

O passo de pesquisa recebe a pilha de arestas P, a ordem das arestas armazenada em h e gera os valores para a função $g : V \to [0, n-1]$. Note que o tamanho do domínio de g é $|V| = cn$. Assim, quanto maior o valor de c mais espaço será necessário para o armazenamento de g.

O passo de pesquisa desempilha de P uma aresta de cada vez. Cada aresta a desempilhada contém um ou mais vértices, os quais não pertencem a nenhuma outra aresta e, conseqüentemente, o valor da função g para tais vértices ainda não foi definido. Seja $N_A = \{v_1, v_2, \ldots, v_j\}$ o conjunto de todos os vértices de a que ainda não foram atribuídos, ou seja $g(v_k) = -1$, sendo $1 \leq k \leq j$. Seja $A = \{u_1, u_2, \ldots, u_i\}$ o conjunto de todos os

[1] O termo **alta probabilidade** é utilizado para significar uma probabilidade tendendo para 1 quando $n \to \infty$

```
MapeamentoOrdenacao (S, G_r, h, P)
  repeat
    for (a = 0; a < n; a++)
      v_1 = h_1(S_a);
      v_2 = h_2(S_a);
          ...
      v_r = h_r(S_a);
      Vertices= {v_1, v_2, ..., v_r};
      Vertices = EliminarSelfLoops (Vertices); /* vide Seção 3.3.3 */
      h(a) = a;
      InserirAresta (a, Vertices, G_r);
  until (not ExisteCiclo (G_r, P));
```

Figura 3.5: Pseudocódigo para os passos de mapeamento e ordenação.

vértices de a que já foram atribuídos. Para cada v_l, sendo $1 \leq l \leq j-1$, atribua 0 para $g(v_l)$ e em seguida faça $g(v_j) = h(a) \oplus \text{Soma}$, onde Soma é uma variável que armazena o ou exclusivo dos valores atribuídos aos vértices em A, ou seja, $\text{Soma} = g(u_1) \oplus g(u_2) \oplus \cdots \oplus g(u_i)$.

A Figura 3.6 ilustra o passo de pesquisa para um 2-grafo G_2 de 6 vértices e 5 arestas. Na Figura 3.6(a) é apresentado o estado inicial do algoritmo, onde o valor -1 indica que nenhum valor foi atribuído a $g(v)$, sendo $0 \leq v \leq 5$. A Figura 3.6(b), ilustra a atribuição da aresta $a = \{5, 3\}$, onde $N_A = \{5, 3\}$ e $A = \emptyset$. Assim, Soma $= 0$, $g(5)$ recebe o valor 0 e $g(3) = h(a) \oplus \text{Soma} = 2 \oplus 0 = 2$. Na Figura 3.6(c), onde a aresta desempilhada é $a = \{4, 3\}$, sendo $N_A = \{4\}$ e $A = \{3\}$, a variável Soma $= g(3) = 2$ e $g(4) = h(a) \oplus \text{Soma} = 1 \oplus 2 = 3$. Na Figura 3.6(d) a aresta desempilhada é $a = \{2, 3\}$, sendo $N_A = \{2\}$ e $A = \{3\}$, o que implica na atribuição $g(2) = h(a) \oplus \text{Soma} = 4 \oplus 2 = 6$. Da mesma forma, na Figura 3.6(e) e (f), $g(1)$ e $g(0)$ recebem os valores 6 e 5, respectivamente.

A Figura 3.7 apresenta um pseudocódigo para o algoritmo do passo de pesquisa. É fácil notar que a complexidade deste código é $O(rn)$, pois para cada aresta desempilhada no laço **while**, seus r vértices são percorridos no laço **for**. Como r é constante, então a complexidade do passo de pesquisa é $O(n)$.

3.3.3 Implementação

Nesta seção apresentamos dois aspectos da implementação realizada para os algoritmos da família MWHC-r, considerando $r = 2, 3$. Primeiramente, descrevemos a representação de uma estrutura de dados, a qual permite a representação de um r-grafo não direcionado

a)

	h(a)=2	h(a)=1	h(a)=4	h(a)=0	h(a)=3
P	{5,3}	{4,3}	{2,3}	{1,2}	{0,2}

	0	1	2	3	4	5	
g	−1	−1	−1	−1	−1	−1	Soma = 0

b)

	h(a)=1	h(a)=4	h(a)=0	h(a)=3
P	{4,3}	{2,3}	{1,2}	{0,2}

	0	1	2	3	4	5	
g	−1	−1	−1	2	−1	0	Soma = 0

c)

	h(a)=4	h(a)=0	h(a)=3
P	{2,3}	{1,2}	{0,2}

	0	1	2	3	4	5	
g	−1	−1	−1	2	3	0	Soma = 2

d)

	h(a)=0	h(a)=3
P	{1,2}	{0,2}

	0	1	2	3	4	5	
g	−1	−1	6	2	3	0	Soma = 2

e)

	h(a)=3
P	{0,2}

	0	1	2	3	4	5	
g	−1	6	6	2	3	0	Soma = 6

f)

P ∅

	0	1	2	3	4	5	
g	5	6	6	2	3	0	Soma = 6

Figura 3.6: Passo de pesquisa para um 2-grafo com 6 vértices e 5 arestas.

```
Pesquisa (P, h, g)
    for (v = 0; v < cn; v++)
        g(v) = −1;
    while (not Vazia (P))
        a = Desempilhar (P);
        Soma = 0;
        for (l = 0; l < r; l++)
            if (g(a[l]) == −1)
                v_j = a[l];
                g(v_j) = 0;
            else Soma = Soma ⊕ g(a[l]);
        g(v_j) = h(a) ⊕ Soma;
```

Figura 3.7: Pseudocódigo para o passo de pesquisa.

sem termos que duplicar as suas arestas. Em seguida, mostramos como gerar um conjunto randômico com r vértices de forma que nenhum par de vértices seja idêntico.

Representação do r-Grafo

A estrutura de dados para representar o r-grafo é baseada na proposta por Ebert em [9]. A estrutura de dados é orientada a arestas, o que implica na representação explícita de cada

aresta do r-grafo. As arestas são armazenadas em um arranjo Arestas que é indexado de 0 a $n-1$. Em cada índice a do arranjo Arestas, são armazenados os r vértices referente a aresta a. As listas de arestas incidentes nos vértices do r-grafo são armazenadas em dois arranjos: Prim e Prox. O elemento Prim$[v]$ define o ponto de entrada para a lista de arestas incidentes no vértice v, enquanto Prox[Prim$[v]$], Prox[Prox[Prim$[v]$]] e assim por diante, definem as arestas subseqüentes que contém v. Nós utilizamos o valor -1 para finalizar a lista. O arranjo Prim deve possuir cn entradas, uma para cada vértice. O arranjo Prox deve possuir rn entradas, pois para cada aresta a, devemos armazenar a na lista de arestas de cada um de seus r vértices.

A Figura 3.8 mostra como um 2-grafo acíclico de 6 vértices e 5 arestas é representado. O grafo é apresentado na Figura 3.8(a) e a sua representação na estrutura de dados é apresentada na Figura 3.8(b). Para descobrir quais são as arestas que contém um determinado vértice v fazemos o seguinte: (i) Percorremos a lista de arestas que inicia em Prim$[v]$ e termina quando Prox[...Prim$[v]$...] $= -1$. (ii) Os valores armazenados nos arranjos Prim e Prox são obtidos pela equação $a + in$, sendo $0 \leq i \leq r-1$. Suponha que $z = a + in$ é um valor armazenado em Prim ou em Prox. Para obtermos a aresta a do valor z, basta tomarmos o valor de z modulo n. Isto é $a = z \bmod n$. Por exemplo, se percorrermos a lista das arestas do vértice 2 obtemos os seguintes valores $\{4, 8, 5\}$ os quais representam as arestas que contém o vértice 2, ou seja, $\{4 \bmod 5 = 4, 8 \bmod 5 = 3, 5 \bmod 5 = 0\}$.

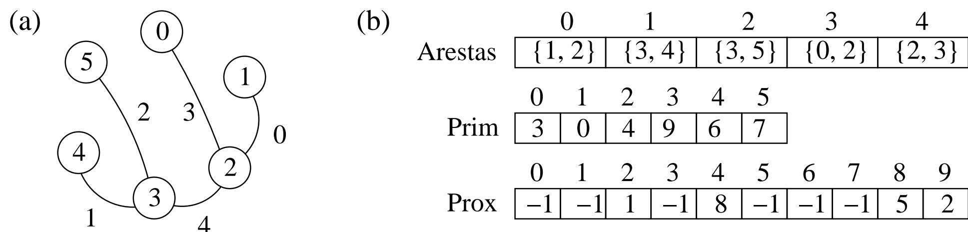

Figura 3.8: (a) 2-grafo com 6 vértices e 5 arestas. (b) Representação do 2-grafo.

A Figura 3.9 apresenta a estrutura de dados para representar um r-grafo. A Figura 3.10 apresenta as funções para criar um r-grafo vazio e para liberar a memória alocada por um r-grafo.

```
typedef struct Aresta {
    TipoInt * Vertices; /* Vertices deve ser alocado para conter r vértices */
}TipoAresta;

typedef struct RGrafo {
   TipoInt NumVertices, n, r;
   TipoAresta * Arestas; /* Arestas deve ser alocado para conter n arestas */
   TipoInt * Prim;       /* Prim deve ser alocado para conter cn entradas */
   TipoInt * Prox;       /* Prox deve ser alocado para conter rn entradas */
}TipoRGrafo;
```

Figura 3.9: Descrição da estrutura de dados para representar um r-grafo.

```
void CriarGrafo (TipoRGrafo * Grafo,TipoInt NumVertices,TipoInt n,TipoInt r) {
    TipoInt i;
    Grafo->Arestas = (TipoAresta*) malloc (n * sizeof(TipoAresta));
    for (i = 0; i < n; i++)
       Grafo->Arestas[i].Vertices = (TipoInt*) malloc (r * sizeof(TipoInt));
    Grafo->Prim = (TipoInt*) malloc (NumVertices * sizeof(TipoInt));
    for (i = 0; i < NumVertices; i++)
      Grafo->Prim[i] = -1;
    Grafo->Prox = (TipoInt*) malloc (r * n * sizeof(TipoInt));
    Grafo->n = n;
    Grafo->r = r;
}
void LiberarGrafo (TipoRGrafo * Grafo) {
    TipoInt i;
    for (i = 0; i < n; i++)
       free (Grafo->Arestas[i].Vertices);
    free (Grafo->Arestas);
    free (Grafo->Prim);
    free (Grafo->Prox);
}
```

Figura 3.10: Funções para criar um r-grafo vazio e para liberar a memória alocada por um r-grafo.

Agora vamos ilustrar como é criado um 2-grafo com 2 arestas e quatro vértices, o qual é mostrado na Figura 3.11(c). Primeiro, vamos definir a seguinte notação $a + in = a \otimes i$, sendo $0 \leq i \leq r - 1$. A Figura 3.11(a) ilustra a representação do 2-grafo ainda vazio. Para inserir a primeira aresta $a = 0$, cujos vértices são $\{0, 1\}$, realizamos os seguintes passos:

1. copiamos os seus vértices para o vetor Aresta[a].Vertices;
2. para cada vértice i em Aresta[a].Vertices, fazemos:
   (a) Prox[$a \otimes i$] = Prim[Aresta[a].Vertices[i]];
   (b) Prim[Aresta[a].Vertices[i]] = $a \otimes i$;

O resultado é mostrado na Figura 3.11(b). Por fim, a aresta $a = 1$, cujos vértices são $\{1, 2\}$, é inserida utilizando o mesmo processo e o resultado é apresentado na Figura 3.11(c). A

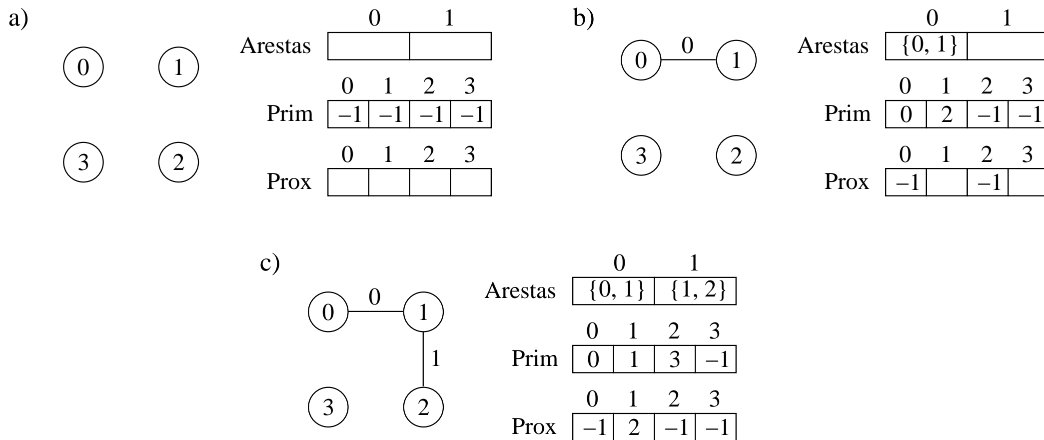

Figura 3.11: Processo de criação de um 2-grafo com 4 vértices e 2 arestas.

Figura 3.12 apresenta a função para inserir em um r-grafo uma aresta a, sendo $0 \leq a \leq n - 1$, cujos vértices estão armazenados na variável Vertices.

Por fim, vamos mostrar como retirar de um r-grafo uma aresta a, sendo $0 \leq a \leq n - 1$. Cada vértice v armazenado em Arestas[a].Vertices possui uma lista de arestas das quais v participa. As listas são simplesmente encadeadas e são implementadas utilizando **cursores**[2]. Assim, para apagar a aresta a, basta remover a aresta de cada uma das listas de seus r vértices. A Figura 3.13 apresenta uma função para realizar a remoção.

[2]Os cursores são números inteiros que representam posições em um arranjo e são utilizados para simular os apontadores da implementação tradicional das listas lineares simplesmente encadeada.

```
void InserirAresta (TipoInt a, TipoInt * Vertices, TipoRGrafo * Grafo) {
    TipoInt i, n = Grafo->n;
    for (i = 0; i < Grafo->r; i++) {
        Grafo->Aresta[a].Vertices[i] = Vertices[i];
        Grafo->Prox[a ⊗ i] = Grafo->Prim[Grafo->Aresta[a].Vertices[i]];
        Grafo->Prim[Grafo->Aresta[a].Vertices[i]] = a ⊗ i;
    }
}
```

Figura 3.12: Função para inserir uma aresta a no r-grafo.

```
void ApagarAresta (TipoInt a, TipoRGrafo * Grafo) {
    TipoInt i, n = Grafo->n;
    TipoInt Prev, Aux;
    for (i = 0; i < Grafo->r; i++) {
        Prev = -1;
        Aux = Grafo->Prim[Grafo->Arestas[a].Vertices[i]];
        while (Aux != a ⊗ i) {
            Prev = Aux;
            Aux = Grafo->Prox[Aux];
        }
        if (Prev == -1)
          Grafo->Prim[Grafo->Arestas[a].Vertices[i]] = Grafo->Prox[a ⊗ i];
        else Grafo->Prox[Prev] = Grafo->Prox[a ⊗ i];
    }
}
```

Figura 3.13: Função para apagar uma aresta a do r-grafo.

O primeiro passo da remoção de uma lista simplesmente encadeada é localizar o elemento anterior ao que se deseja remover. Esta operação deve ser efetuada com complexidade $O(1)$ para que o teste de aciclicidade possa ser realizado com complexidade $O(n)$. Assim, uma opção para permitir a remoção de uma lista linear simplesmente encadeada com complexidade $O(1)$ é criar um outro arranjo Prev com rn entradas. Para uma aresta a, sendo $0 \leq a \leq n-1$, Prev[$a \otimes i$] (para $i = 0, 1 \ldots, r-1$) armazena a aresta que precede $a \otimes i$. Assim, se Prox[$a \otimes i$] $= b \otimes j$ então Prev[$b \otimes j$] $= a \otimes i$. Para todas arestas a em que Prim[v] $= a \otimes i$, onde $v \in$ Arestas[a].Vertices, Prev[$a \otimes i$] = -1. No entanto, isso não é necessário, uma vez que o tamanho das listas pode ser considerado constante. Isso porque o maior grau de um vértice em um r-grafo randômico é $O(\log|V|/\log\log|V|)$ (vide Molloy, Reed [24] para detalhes).

Impedindo a Ocorrência de *Self-loops*

Nesta seção apresentamos como evitar *self-loops* para $r = 2, 3$. Um algoritmo para valores genéricos de r é apresentado em [2]. Nós assumimos que cada função h_i, sendo $1 \leq i \leq r$, retorna um valor randômico no intervalo $[0, |V| - 1]$.

Seja $\{v_1, v_2, \ldots, v_r\}$ o conjunto de vértices de uma certa aresta a. Para $r = 2$, utilizamos o seguinte artifício para eliminar *self-loops*: $v_2 = (v_1 + v_2 + 1) \bmod |V|$. Isto gera qualquer par de vértices distintos entre 0 e $|V| - 1$ com igual probabilidade. Para $r = 3$, utilizamos o seguinte:

```
v2 = v1 + v2 + 1;
v3 = v1 + v3 + 1;
if (v3 >= v2) v3 = v3 + 1;
v2 = v2 mod |V|;
v3 = v3 mod |V|;
```

O segundo vértice recebe aleatoriamente qualquer valor no intervalo $[0, |V| - 1]$, exceto o valor atribuído ao vértice v_1. O terceiro vértice também recebe aleatoriamente qualquer valor no intervalo $[0, |V| - 1]$, exceto os valores atribuídos aos vértices v_1 e v_2. Esta solução garante que qualquer uma das triplas distintas possam ser geradas com igual probabilidade. Como cada função h_i é computada em tempo constante, o tempo necessário para gerar todos os vértices de uma aresta é $O(r)$.

3.4 O Algoritmo FCH

Nesta seção apresentamos o algoritmo FCH [14]. O algoritmo FCH é o único da literatura capaz de gerar uma FHPM de ordem não preservada, cujo espaço necessário para seu armazenamento é próximo do limite inferior determinado por Mehlhorn [23]. O algoritmo FCH também é baseado na abordagem MOS, tendo passos explícitos de mapeamento, ordenação e pesquisa, os quais são ilustrados na Figura 3.14 e descritos nas seções 3.4.1, 3.4.2 e 3.4.3, respectivamente.

O algoritmo FCH difere radicalmente dos algoritmos da família MWHC-r apresentada na seção 3.3. As principais diferenças são: (i) nenhum grafo é gerado pelo algoritmo e (ii) a FHPM gerada não é de ordem preservada.

O algoritmo FCH procura por uma função $g : [0, b - 1] \to [0, n - 1]$, onde $b = |B|$, de forma que a função $h : S \to [0, n - 1]$ definida como

$$h(s) = (h_{20}(s) + g(i)) \bmod n \tag{3.4}$$

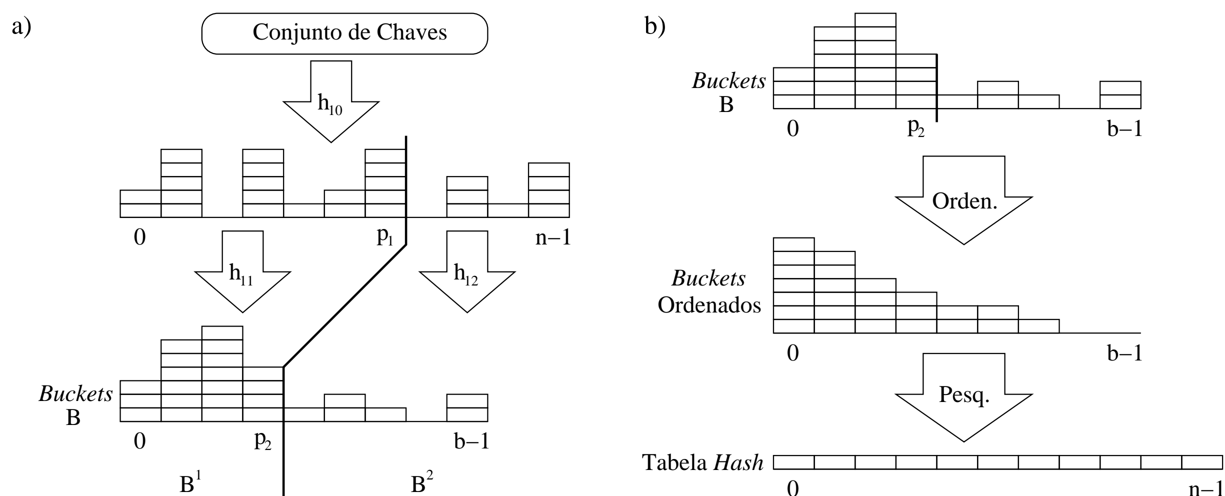

Figura 3.14: (a) Passo de mapeamento. (b) Passos de ordenação e pesquisa.

seja uma bijeção. No passo de mapeamento, as chaves de S são mapeadas para um conjunto de *Buckets* $B = \{B_1, B_2, \ldots, B_{b-1}\}$. No passo de ordenação, os *Buckets* em B são ordenados de forma não crescente pelos seus respectivos tamanhos $|B_i|$, onde $0 \leq i \leq b-1$. No passo de pesquisa, as chaves de cada B_i são distribuídas nos registros desocupados da tabela *hash* enquanto são definidos os valores da função g. O parâmetro b é determinado da seguinte forma:

$$b = |B| = \left\lceil \frac{\beta n}{\log n + 1} \right\rceil, \ \ 2 \leq \beta \leq 4 \tag{3.5}$$

Observando a Eq. (3.5), notamos que o intervalo de mapeamento das chaves de S foi reduzido de $[0, n-1]$ para $[0, b-1]$. Esta é a característica que permite a geração de uma FHPM que pode ser armazenada com complexidade $O(n)$ *bits* (considerando que o tamanho de cada palavra do computador é $\log n$), pois b é o tamanho do domínio da função g. Já a função $h_{20} : S \to [0, n-1]$, é uma função *hash* que obrigatoriamente deve ser perfeita para as chaves de $B_i \in B$, ou seja, as chaves de um certo *Bucket* devem ser mapeadas em inteiros distintos no intervalo $[0, n-1]$. No entanto, a função h_{20} não precisa ser perfeita para todas as chaves de S.

3.4.1 Mapeamento

Os objetivos do passo de mapeamento são mapear o conjunto S para números inteiros no intervalo $[0, n-1]$, comprimir o intervalo de mapeamento de $[0, n-1]$ para $[0, b-1]$ e

separar grandes grupos de chaves de pequenos grupos. O passo de mapeamento é ilustrado na Figura 3.14(a). Ele inicia com a utilização de uma função *hash* não perfeita $h_{10} : S \to [0, n-1]$ para mapear as n chaves de S para o intervalo $[0, n-1]$. Em seguida, h_{10} é composta com duas funções $h_{11} : [0, p_1 - 1] \to [0, p_2 - 1]$ e $h_{12} : [p_1, n-1] \to [p_2, b-1]$ para determinar o *Bucket* B_i para o qual uma determinada chave $s \in S$ é mapeada. A composição é feita das seguinte forma:

$$i = \begin{cases} h_{10} \circ h_{11} & \text{se } h_{10}(s) < p_1, \\ h_{10} \circ h_{12} & \text{caso contrário} \end{cases}$$

O conjunto de *Buckets* B é constituído de dois subconjuntos B^1 e B^2, sendo que h_{11} e h_{12} determinam as chaves de S que fazem parte de B^1 e B^2, respectivamente. As funções h_{11} e h_{12} dependem de dois parâmetros p_1 e p_2, respectivamente. Bons valores para os dois parâmetros são $p_1 = 0.6n$ e $p_2 = 0.3b$. Estes valores foram determinados experimentalmente. Isto significa que 60% das chaves serão mapeadas para 30% dos *Buckets*. O objetivo é fazer com que os *Buckets* do subconjunto B^1 tenham muitas chaves. A idéia é fazer com que as suas chaves sejam distribuídas na tabela *hash* mais cedo durante o passo de pesquisa, uma vez que a tabela estará mais vazia proporcionando uma maior probabilidade de sucesso.

Por outro lado, os outros 40% das chaves são espalhados por h_{12} em 70% dos *Buckets*. O objetivo é fazer com que os *Buckets* do subconjunto B^2 tenham poucas chaves. É desejável o processamento de *Buckets* menores no fim do passo de pesquisa, pois isto aumenta a probabilidade de obtermos sucesso na distribuição de suas chaves na tabela *hash*.

A Figura 3.15 apresenta um pseudocódigo para o passo de mapeamento do algoritmo FCH. O algoritmo recebe o conjunto de chaves S como entrada e produz o conjunto de *Buckets* B e o tamanho $\max_{|B_i|}$ do maior *Bucket* B_i em B. A complexidade do passo de mapeamento é claramente linear no número de chaves em S, pois o conjunto S é percorrido somente uma vez para formar os *Buckets*.

3.4.2 Ordenação

O objetivo do passo de ordenação é ordenar o conjunto B de forma não crescente pelo tamanho de cada $B_i \in B$. Isto garante que *Buckets* maiores serão processados primeiro no passo de pesquisa. O passo de ordenação é ilustrado na Figura 3.14(b).

Devido a forma como os *Buckets* são formados no passo de mapeamento, podemos determinar o número de *Buckets* que possuem um certo tamanho. Com esta informação,

```
Mapeamento (S, B, max_{|B_i|})
    b = ⌈βn/(log n + 1)⌉;
    p_1 = 0.6n;
    p_2 = 0.3n;
    for (j = 0; j < n; j++)
        i = h_10(s);
        if (i < p_1)  i = h_11(i);
        else i = h_12(i);
        Inserir (s, B_i);
        max_{|B_i|} = max(|B_i|, max_{|B_i|});
```

Figura 3.15: Pseudocódigo para o passo de mapeamento.

o passo de ordenação é implementado com complexidade linear no número de *Buckets*. Para isto utilizamos a estrutura de dados mostrada na Figura 3.16. Para cada linha L, sendo $0 \leq L \leq \max_{|B_i|}$, existe um apontador para uma lista de *Buckets* em B que possuem tamanho L. O maior *Bucket* $\max_{|B_i|}$ é determinado no passo de mapeamento e o número de elementos em cada lista é armazenado no índice zero das listas.

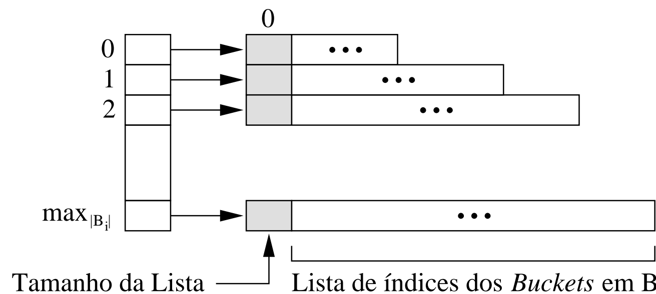

Figura 3.16: Estrutura de dados auxiliar para o passo de ordenação.

A construção da estrutura de dados é realizada com o seguinte algoritmo:

1. Determinar quantos *Buckets* em B possuem tamanho L, sendo $0 \leq L \leq \max_{|B_i|}$. Em seguida, armazenar este valor no índice zero de cada lista.

2. Para cada $B_i \in B$, se $|B_i| = L$, então armazenamos i na lista da linha L.

Para termos a seqüência de *Buckets* ordenada não crescentemente, basta percorrer as listas na ordem decrescente dos tamanhos possíveis para os *Buckets*. Ou seja, a partir da lista que se encontra na linha $\max_{|B_i|}$ até a lista da linha 1. A lista da linha 0 deve ser desconsiderada,

uma vez que os *Buckets* de tamanho zero não precisam ser processados, pois não possuem nenhuma chave.

A Figura 3.17 apresenta um pseudocódigo para o passo de ordenação do algoritmo FCH. O algoritmo recebe o conjunto de *Buckets* B não ordenado e retorna na variável BucketsOrdenados a estrutura de dados mostrada na Figura 3.16, a qual representa a ordenação do conjunto B. A variável NumBuckets_L representa o número de *Buckets* de tamanho L, sendo $0 \leq L \leq \max_{|B_i|}$. A complexidade do passo de ordenação é linear em $|B| = b$, pois o conjunto B é percorrido duas vezes durante a sua ordenação.

```
Ordenacao (max_|Bi| , B , BucketsOrdenados)
   for (L = 0; L ≤ max_|Bi| ; L++)
      BucketsOrdenados[L][0] = 0;
      NumBuckets_L[L] = 0;
   for (i = 0; i < b; i++)
      BucketsOrdenados[|Bi|][0]++;
      NumBuckets_L[|Bi|]++;
   for (i = 0; i < b; i++)
      if (NumBuckets_L[|Bi|] > 0)
         BucketsOrdenados[|Bi|][NumBuckets_L[|Bi|]] = i;
         NumBuckets_L[|Bi|] --;
```

Figura 3.17: Pseudocódigo para o passo de ordenação.

3.4.3 Pesquisa

O passo de pesquisa recebe a seqüência de *Buckets* ordenada como entrada e determina os valores da função g para cada *Bucket* em B, os quais fazem da função apresentada na Eq. (3.4) uma FHPM. A função gerada tem uma forma simples e é facilmente computável para as chaves de S.

A primeira etapa do passo de mapeamento é escolher uma função h_{20} aleatoriamente de uma família de funções possíveis, a qual deve mapear as chaves em cada B_i para valores distintos no intervalo $[0, n-1]$. Fox, Chen and Heath [14] utilizam o termo padrão P_i para denotar os valores de h_{20} correspondentes as chaves em B_i. Em seguida, os valores da função g para cada B_i são selecionados através do alinhamento de um valor arbitrário interno a cada P_i com as posições vazias da tabela *hash*. O processo pára quando o alinhamento permite que os valores de P_i sejam simultaneamente distribuídos pela função h em endereços vazios da tabela.

Sejam x o endereço na tabela *hash* de um registro ainda não ocupado e u o membro de um padrão P_i a ser alinhado com x. O alinhamento de x com u que determina o valor de g para um certo B_i se dá da seguinte forma:

$$g(i) = (n + x - u) \bmod n$$

O valor de x é rotacionado através dos endereços desocupados na tabela *hash* até que todos os valores de P_i sejam mapeados sem colisões. Caso x retorne a seu valor inicial, o algoritmo é reiniciado no passo de pesquisa. Caso o passo de pesquisa seja reiniciado n vezes, o conjunto de *Buckets* é abandonado e o algoritmo é reiniciado no passo de mapeamento.

Segundo Fox, Chen e Heath [14], o número de tentativas N_t para distribuir um padrão P_i em uma tabela *hash* com f entradas ocupadas é aproximadamente

$$N_t \simeq \frac{\binom{n}{|P_i|}}{\binom{n-f}{|P_i|}}$$

Utilizando esta equação podemos evitar iterações desnecessárias no passo de pesquisa. Isto porque, se $N_t > n$, é pouco provável que o padrão P_i possa ser distribuído na tabela *hash*, independente da função h_{20} utilizada. Nestes casos, o algoritmo é reiniciado no passo de mapeamento.

A Figura 3.18 apresenta uma estrutura de índice que melhora consideravelmente a eficiência do passo de pesquisa. A estrutura é composta por três tabelas de tamanho n denominadas tabela randômica (TR), tabela de mapeamento (TM) e a tabela *hash* (TH). A tabela randômica é utilizada para relembrar quais endereços da tabela *hash* estão ocupados e quais estão vazios. Inicialmente, TR contém uma permutação randômica dos endereços ($[0, n-1]$) de TH. O apontador EndOcupados é iniciado com zero. Ele é um invariante, pois os endereços armazenados à sua esquerda estão ocupados e os armazenados à sua direita (inclusive) estão desocupados. Isto garante que somente endereços não utilizados são pesquisados para que um padrão P_i seja distribuído na tabela *hash*.

Para qualquer endereço x da tabela *hash*, a tabela de mapeamento contém apontadores para a tabela randômica de forma que $\text{TR}[\text{TM}[x]] = x$. Assim, dado um endereço x de um registro vazio na tabela *hash*, podemos localizar sua posição na tabela randômica através da tabela de mapeamento ($\text{TM}[x] = y$). Suponha que na Figura 3.18(a) o endereço x, o qual está armazenado na posição y da tabela randômica, foi selecionado para ser ocupado e o invariante precisa ser mantido. Para isto, basta trocarmos $\text{TR}[\text{EndOcupados}] = w$ com $\text{TR}[y] = x$ e em seguida atribuir $\text{EndOcupados} = z$ para $\text{TM}[\text{TR}[\text{EndOcupados}]]$ e y para $\text{TM}[\text{TR}[y]]$. Por fim, o apontador EndOcupados é incrementado de um. O resultado deste

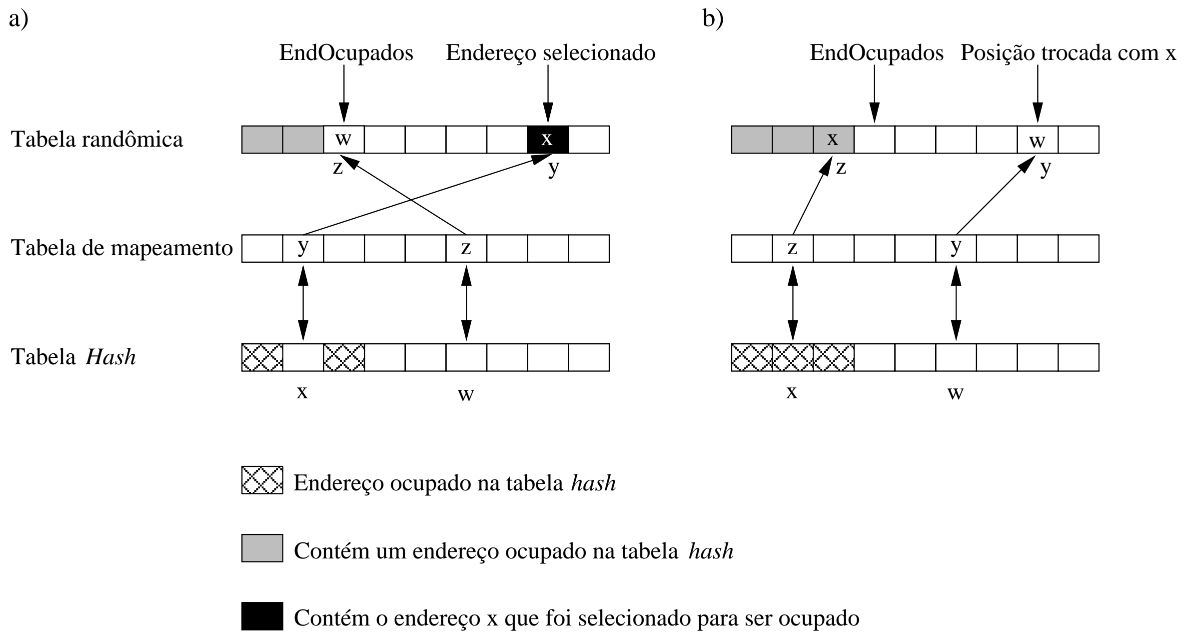

Figura 3.18: Estrutura de índice para o preenchimento de uma posição vazia da tabela *hash*.

processo é ilustrado na Figura 3.18(b). Quando $|P_i| > 1$, é necessário uma seqüência de trocas.

A Figura 3.19 apresenta um pseudocódigo para o passo de pesquisa do algoritmo FCH. O algoritmo recebe o conjunto de *Buckets* B, a estrutura de dados BucketsOrdenados que representa a ordenação não ascendente de B, o maior tamanho $\max_{|B_i|}$ de um *Bucket* e obtém as funções h_{20} e g que fazem da Eq. (3.4) uma FHPM. Os autores do algoritmo FCH não apresentaram nenhuma análise da complexidade do passo de pesquisa, no entanto, sua complexidade é exponencial, pois o tamanho do maior *Bucket* é $O(\log n)$. Assim, para grandes valores de n o número de iterações para obter uma função h_{20} perfeita para todos os *Buckets* tende ao infinito.

```
Pesquisa (B, BucketsOrdenados, max_|Bi|, h20, g)
  Inicializar TR com uma permutação randômica de valores no intervalo [0, n − 1];
  repeat
    Iteracoes = 0;
    repeat
      Gerar a função h20;
    until h20 ser perfeita para cada Bi;
    Colisao = false;
    for(L = max_|Bi|; (L > 0) && (not Colisao); L−−)
      for(k = 1; (k <= BucketsOrdenados[L][0]) && (not Colisao); k++)
        Colisao = true;
        i = BucketsOrdenados[L][k];
        Calcular Nt para Bi;
        if (Nt > n) Reiniciar passo de mapeamento;
        for(d = 0; (d < (n − EndOcupados)) && Colisao; d++)
          NumChavesDistribuidas = 0;
          Colisao = false;
          j = IndiceDeAlinhamento;
          s = Bi[j];
          x = TR[EndOcupados + d];
          u = h20(s);
          g(i) = (n + x − u) mod n;
          repeat
            x = (h20(s) + g(i)) mod n;
            y = TM[x];
            if (y >= EndOcupados)
              TR[y] = TR[EndOcupados];
              TR[EndOcupados] = x;
              TM[TR[EndOcupados]] = EndOcupados;
              TM[TR[y]] = y;
              EndOcupados++;
              NumChavesDistribuidas ++;
            else
              Colisao = true;
              EndOcupados = EndOcupados − NumChavesDistribuidas;
              break;
            j = (j + 1) mod |Bi|;
            s = Bi[j];
          until ((j mod |Bi|) == IndiceDeAlinhamento);
    Iteracoes++;
    if (Iteracoes > n) Reiniciar passo de mapeamento;
  until not Colisao;
```

Figura 3.19: Pseudocódigo para o passo de pesquisa.

Capítulo 4

Resultados Experimentais

Neste capítulo apresentamos os conjuntos de chaves utilizados nos experimentos e os resultados do estudo comparativo. O ambiente de execução dos experimentos é composto por uma máquina pentium IV 2.4 GHz executando sistema operacional Linux 2.6.7 e com 4GB de memória principal. Todos os resultados apresentados são médias sobre 50 execuções dos algoritmos e todos os tempos estão em segundos. A Tabela 4.1 resume a simbologia e as abreviações utilizadas na apresentação dos resultados. Nos experimentos realizados utilizamos os valores sugeridos pelos autores dos algoritmos para as constantes c e β definidas nas Seções 3.3.1 e 3.4, respectivamente. Os valores são $c = 2.09$ para o algoritmo MWHC-2, $c = 1.23$ para o algoritmo MWHC-3 e $\beta = 3.5$ para o algoritmo FCH.

Símbolo	Significado
N_B	Número de *bytes* para representar um número inteiro na arquitetura considerada.
N_C	Número de campos de cada registro da tabela *hash*.
N_{AL}	Número médio de alinhamentos para distribuir um padrão P_i na tabela.
NIh_{20}	Número de iterações para obter uma função h_{20} perfeita para os *Buckets*.
N_i	Número de iterações para gerar um r-grafo acíclico, sendo $r = 2, 3$.
Map.	Abreviação para passo de mapeamento.
Ord.	Abreviação para passo de Ordenação.
Pes.	Abreviação para passo de Pesquisa.
hu	Função *hash* Universal.
hz	Função *hash* de Zobrist.
hj	Função *hash* de Jenkins.
hl	*Hashing* linear.
DP	Desvio Padrão.

Tabela 4.1: Simbologia e abreviações utilizadas na apresentação dos resultados.

4.1 Conjunto de Chaves

Os conjuntos de chaves S utilizados nos experimentos foram o vocabulário extraído da coleção TREC-VLC2 (Very Large Collection 2) [18] e um conjunto de URLs coletadas da web. A Tabela 4.2 apresenta as principais características dos conjuntos de chaves. Os tamanhos da menor chave, da maior chave e tamanho médio das chaves são dados em número de caracteres.

S	n	Menor Chave	Maior Chave	Tamanho médio das chaves
TREC-VLC2	10,935,928	1	25	8.52
URLs	10,935,928	7	493	57.23

Tabela 4.2: Propriedades dos conjuntos de chaves utilizados nos experimentos.

4.2 Impacto das Funções *Hash* no Método Endereçamento Aberto

Nesta seção analisamos o impacto do fator de carga $\alpha = n/m$ no processo de resolução de colisões por *hashing* linear. Além disso, discutimos também o impacto das três funções *hash* apresentadas no Capítulo 2 no desempenho do método endereçamento aberto que resolve colisões por *hashing* linear.

Uma função *hash* é utilizada para mapear um conjunto S com n chaves para um certo intervalo $[0, m-1]$, onde m é o tamanho da tabela *hash*. Quanto mais próximo o valor de m estiver de n, maior será o fator de carga e maior será a probabilidade de acontecer colisões. Conseqüentemente, o tempo para se pesquisar uma chave em S é degradado com o aumento do fator de carga (vide Seção 3.2). Para comprovar isso, realizamos o seguinte experimento:

1. Para um certo valor de α, construímos uma tabela *hash* com todas as chaves da coleção TREC-VLC2 e medimos o número médio de colisões por chave (Colisões/n).

2. Em seguida, medimos o tempo para pesquisar todas as chaves do conjunto S na tabela quando são utilizadas as funções *hash* universal (hu), de Zobrist (hz) e de Jenkins (hj) no *hashing* linear. As chaves foram pesquisadas na ordem em que elas aparecem no conjunto S.

A Tabela 4.3 apresenta os resultados obtidos variando o fator de carga de 70% a 25%. Como esperado, as funções hu e hz apresentam aproximadamente a mesma taxa de colisão (Colisões/n), onde n é o número de chaves. Porém, a função hu é mais lenta, pois além de realizar a mesma quantidade de adições da função hz, ela também realiza um acréscimo de multiplicações igual ao tamanho da chave pesquisada. Já a função hj apresenta uma taxa de colisões mais alta do que as demais, no entanto, ela é mais rápida (veja na Tabela 4.3 o tempo de cada função quando Colisões/$n = 0.5$) e necessita de somente um número inteiro para ser representada internamente. A maior desvantagem da função hj é que ela necessita de um fator de carga menor para ter o mesmo desempenho das funções hu e hz, o que provoca um maior desperdício de espaço.

	hu		hz		hj	
α	Colisões/n	Tempo (s)	Colisões/n	Tempo (s)	Colisões/n	Tempo (s)
70%	1.16	7.72	1.17	7.34	88.76	46,174.42
65%	0.92	7.21	0.93	6.86	0.94	7.00
60%	0.76	6.87	0.75	6.56	0.88	6.86
55%	0.62	6.63	0.61	6.23	0.81	6.72
50%	0.50	6.38	0.50	6.03	0.73	6.59
45%	0.41	6.18	0.41	5.84	0.62	6.35
40%	0.34	6.06	0.33	5.69	0.50	6.01
35%	0.27	5.93	0.27	5.55	0.34	5.69
30%	0.21	5.80	0.21	5.44	0.23	5.56
25%	0.17	5.74	0.17	5.38	0.20	5.50

Tabela 4.3: Influência do fator de carga no tempo para pesquisar todas as chaves do conjunto extraído da coleção TREC-VLC2 considerando as funções hu, hz e hj.

Um outro fator que influencia no tempo de pesquisa na tabela é o tamanho das chaves pesquisadas. Isso porque a complexidade para computar o valor das funções *hash* para uma dada chave $s \in S$ é $O(|s|)$, onde $|s|$ é o número de caracteres da chave s. Dessa forma, realizamos o mesmo experimento descrito anteriormente para a coleção de URLs, a qual possui um tamanho médio das chaves maior do que a coleção TREC-VLC2.

A Tabela 4.4 apresenta os resultados obtidos. Para uma dada chave $s \in S$, a função de Jenkins executa aproximadamente $6|s|$ instruções enquanto as funções de Zobrist e Universal executam aproximadamente $10|s|$ e $20|s|$, respectivamente. Assim, mesmo apresentando uma maior taxa de colisão, com o aumento do tamanho médio das chaves e fatores de carga abaixo de 65%, a função de Jenkins é mais eficiente do que as funções de Zobrist e Universal, pois executa 60% e 30% das instruções executadas pelas funções hz e hu, respectivamente.

Um fato intrigante é que mesmo executando 50% das instruções executadas pela função *hash* Universal, a função de Zobrist é menos eficiente. Como a maior URL possui 493 caracteres, para armazenar o conjunto de pesos da função de Zobrist são necessários aproximadamente 491.1KB e para armazenar o conjunto de pesos da função *hash* Universal são necessários aproximadamente 1.93KB. Devido a esta diferença, o acesso ao conjunto de pesos da função de Zobrist é mais caro e provoca *cache misses* gerando latência de ciclos de CPU e diminuindo a eficiência da função. Isto não ocorreu para a coleção TREC-VLC2, pois a maior chave possui 25 caracteres e os tamanhos dos conjuntos de pesos das funções de Zobrist e Universal são aproximadamente 25KB e 100 *bytes*, respectivamente. Ou seja, são pequenos o suficiente para serem totalmente armazenados na memória *cache*.

	hu		hz		hj	
α	Colisões/n	Tempo (s)	Colisões/n	Tempo (s)	Colisões/n	Tempo (s)
70%	1.17	14.37	1.16	14.87	381.69	85434.55
65%	0.93	13.63	0.93	14.14	0.94	12.00
60%	0.75	13.06	0.75	13.56	0.88	11.84
55%	0.61	12.62	0.61	13.10	0.81	11.65
50%	0.50	12.28	0.50	12.76	0.73	11.34
45%	0.41	11.98	0.41	12.43	0.62	11.01
40%	0.33	11.74	0.33	12.21	0.50	10.61
35%	0.27	11.56	0.27	12.00	0.34	10.03
30%	0.21	11.41	0.21	11.85	0.23	9.77
25%	0.17	11.26	0.17	11.72	0.20	9.68

Tabela 4.4: Influência do fator de carga no tempo para pesquisar todas as chaves do conjunto extraído da coleção de URLs considerando as funções hu, hz e hj.

Uma outro fato que desperta curiosidade é a explosão que ocorre com a taxa de colisão e com o tempo da função de Jenkins para $\alpha = 70\%$ nas duas coleções. As colisões podem ser classificadas em colisões primárias e secundárias. As colisões primárias ocorrem quando o valor de j na Eq. (1.1) é igual a 1 e as colisões secundárias ocorrem quando temos $j \geq 2$. A causa desta explosão é o grande número de colisões secundárias, o que gera listas lineares longas para serem pesquisadas. O mesmo ocorre para as funções de Zobrist e Universal para $\alpha > 99\%$. O fenômeno ocorre tão cedo para a função de Jenkins devido ao fato dela não possuir probabilidade igual a $1/m$ de ocorrência de colisão.

Independente da função *hash* utilizada, podemos observar claramente que a taxa de colisão decresce com a diminuição do fator de carga (aumento do tamanho da tabela *hash*). Assim, o método endereçamento aberto com tratamento de colisões por *hashing* linear é

mais um exemplo na computação onde podemos trocar espaço por tempo. Porém, existem certas aplicações que, devido ao tamanho do conjunto de chaves, a utilização deste método se torna inviável. Nas próximas seções mostraremos que a utilização de FHPMs é uma alternativa viável, pois requer menos espaço e o desempenho é praticamente o mesmo.

4.3 Hashing Perfeito Versus Hashing Linear

Nesta seção comparamos a utilização de FHPMs geradas pelos algoritmos FCH, MWHC-2 e MWHC-3 com o *hashing* linear. As métricas utilizadas na comparação são:

1. tempo para computar o endereço de armazenamento ou recuperação de uma chave na tabela e

2. quantidade de memória interna consumida pelo sistema de busca para armazenar a tabela *hash* e a função que a indexa.

A Tabela 4.5 apresenta os resultados de tempo para se pesquisar todas as chaves extraídas da coleção TREC-VLC2 na tabela *hash* quando utilizamos as FHPMs consideradas e o *hashing* linear para endereçar a tabela. Medimos o impacto da utilização de cada uma das funções hu, hz e hj para substituir as funções que compõe as FHPMs ou o *hashing* linear, cujo domínio é o conjunto S. Por exemplo, substituindo as funções h_{10} e h_{20} que compõem as FHPMs geradas pelo algoritmo FCH, ou substituindo as funções h_1, h_2 e h_3 que compõem as FHPMs geradas pelos algoritmos MWHC-2 e MWHC-3.

A FHPM que apresenta o melhor desempenho é a gerada pelo algoritmo MWHC-2 sendo composta pela função hj. O mecanismo de busca utilizando essa FHPM só é superado em eficiência pelo que utiliza *hashing* linear com α menor que 35% (inclusive). Para o *hashing* linear é preferível a utilização da função hz ao invés da função hj, pois hz possui uma taxa de colisão menor. Uma vez que para FHPMs não há colisões, a função hj apresenta o melhor desempenho quando usada para compor as FHPMs, pois é mais rápida do que as funções hu e hz.

A razão pela qual o *hashing* linear é mais rápido do que as FHPMs geradas pelos algoritmos FCH, MWHC-2 e MWHC-3 para fatores de carga menores do que 65%, 35% e 55%, respectivamente, é o fato das FHPMs serem mais dispendiosas computacionalmente. As tabelas de pesos de números pseudo-aleatórios das funções hu e hz assim como a semente inicial da função hj são pequenas o suficiente para serem totalmente armazenadas em *cache*. Logo, a latência no acesso à memória para computar as funções hu, hz e hj é

Algoritmos	Funções	α	hu	hz	hj
FCH	$h(s) = (h_{20}(s) + g(i)) \bmod n$	100%	7.87	7.56	7.04
hl	$h_j = (h(s) + j) \bmod m,\ \text{para } 1 \leq j \leq m-1$	65%	7.21	6.86	7.00
MWHC-2	$h(s) = g(h_1(s)) \oplus g(h_2(s))$	100%	6.40	5.91	5.57
hl	$h_j = (h(s) + j) \bmod m,\ \text{para } 1 \leq j \leq m-1$	35%	5.93	5.55	5.69
MWHC-3	$h(s) = g(h_1(s)) \oplus g(h_2(s)) \oplus g(h_3(s))$	100%	7.63	6.99	6.25
hl	$h_j = (h(s) + j) \bmod m,\ \text{para } 1 \leq j \leq m-1$	55%	6.63	6.23	6.72

Tabela 4.5: Comparação da utilização de FHPMs versus *hashing* linear considerando o tempo para pesquisar todas as chaves do conjunto extraído da coleção TREC-VLC2, variando as funções hu, hz e hj na composição das FHPMs e do *hashing* linear.

mínima. A única latência que ocorre no *hashing* linear é no acesso à tabela *hash* para tratar colisões. Como para valores pequenos de α poucas colisões ocorrem e quase sempre o valor de $j = 0$, para computar o endereço na tabela *hash* de uma dada chave s, realizamos por chave pouco mais de um acesso à tabela, em média.

Já para as FHPMs dos algoritmos FCH, MWHC-2 e MWHC-3 um único acesso é necessário para verificar se a chave pesquisada está presente na tabela. No entanto, o custo maior é no acesso ao arranjo g. Este arranjo é muito grande para ser armazenado em *cache* e como ele é acessado aleatoriamente, *cache misses* irão ocorrer, o que, além do acesso à tabela *hash*, também provoca latência, diminuindo a eficiência das FHPMs. A FHPM gerada pelo algoritmo FCH é a menos eficiente, pois além do acesso ao arranjo g ela realiza 3 operações de $\bmod$ (para as funções h_{10}, h_{20} e h_{11} ou h_{12}) e as demais FHPMs não utilizam esta operação que é cara em termos computacionais.

Agora vamos comparar o *overhead* de espaço de cada um dos métodos considerando a coleção TREC-VLC2. O *overhead* de espaço proporcionado pelas FHPMs corresponde à quantidade de memória necessária para armazenar a função g. Já para o *hashing* linear, corresponde a quantidade de entradas a mais na tabela para obter o mesmo desempenho das FHPMs. Vamos supor que cada entrada na tabela *hash* contenha apenas um apontador ($N_C = 1$) para a respectiva chave no conjunto de chaves (vide Figura 3.1) e que cada número inteiro ou apontador é representado por 4 *bytes* ($N_B = 4$).

A Tabela 4.6 apresenta o *overhead* de espaço para cada FHPM considerada e para o *hashing* linear com fator de carga que o torna equivalente (em termos de eficiência) à FHPM logo acima. A FHPM gerada pelo algoritmo FCH é a única que é melhor em termos de eficiência e em termos de espaço. No entanto, não é comum que em cada entrada da tabela só exista um apontador, pois na prática são freqüentes as situações em que necessitamos

de mais campos para representar outras informações. Logo, basta termos $N_C \geq 2$ para que as outras duas FHPMs sejam mais vantajosas em termos de tempo e espaço.

Algoritmos	α	m	Tamanho do domínio da função g	*Overhead*	*Overhead* para $n = 10,935,928$
FCH	100%	n	$\left\lceil \frac{3.5n}{(\log n+1)} \right\rceil$	$N_B \times \left\lceil \frac{3.5n}{(\log n+1)} \right\rceil$	6.1MB
hl	65%	$\lceil \frac{20n}{13} \rceil$	0	$N_B \times \lceil N_C \times \frac{7n}{13} \rceil$	22.5MB
MWHC-2	100%	n	$2.09n$	$N_B \times \lceil 2.09n \rceil$	87.1MB
hl	35%	$\lceil \frac{20n}{7} \rceil$	0	$N_B \times \lceil N_C \times \frac{13n}{7} \rceil$	77.5MB
MWHC-3	100%	n	$1.23n$	$N_B \times \lceil 1.23n \rceil$	51.3MB
hl	55%	$\lceil \frac{20n}{11} \rceil$	0	$N_B \times \lceil N_C \times \frac{9n}{11} \rceil$	34.2MB

Tabela 4.6: Comparação da utilização de FHPMs versus *hashing* linear considerando o *overhead* de espaço para armazenar a tabela *hash* e o arranjo g de cada FHPM para a coleção TREC-VLC2.

Os tamanhos das chaves no conjunto S também influenciam no tempo de avaliação tanto do *hashing* linear quanto das FHPMs. Para verificar como se comporta cada um dos métodos para tamanhos de chaves maiores realizamos as mesmas medidas apresentadas anteriormente para a coleção de URLs. A Tabela 4.7 apresenta os resultados de tempo para se pesquisar todas URLs na tabela *hash* variando as funções hu, hz e hj na composição das FHPMs consideradas e do *hashing* linear com fator de carga que o torna equivalente, em termos de eficiência, à FHPM logo acima.

Como as URLs possuem um tamanho médio muito maior do que as chaves da coleção TREC-VLC2 e a complexidade das funções hu, hz e hj é $O(|s|)$, então, quanto mais funções *hash* forem computadas para avaliar a FHPM pior é o seu desempenho. Assim, a FHPM que apresentou o pior resultado foi a do algoritmo MWHC-3, pois é composta por três funções *hash*. A utilização das funções hu e hz tornam as FHPMs ineficientes, pois para computá-las as chaves são processadas *byte* a *byte*. Apesar de executar menos instruções do que a função hu, a utilização da função hz para compor as FHPMs implica em um pior tempo de avaliação, pois o acesso à seu conjunto de pesos provoca latência de ciclos de CPU (vide Seção 4.2).

A FHPM que apresenta o melhor desempenho para a coleção de URLs é a do algoritmo MWHC-2 sendo composta pela função hj. O mecanismo de busca utilizando essa FHPM só é superado em eficiência pelo que utiliza *hashing* linear com α menor que 40% (inclusive). Diferentemente do que acontece para a coleção da TREC-VLC2, para o *hashing* linear é preferível a utilização da função hj ao invés da função hz, pois a diferença de eficiência

causada pelo fato das URLs terem tamanho médio quase 8 vezes maior compensa a maior taxa de colisões causada pela função hj.

Algoritmos	Funções	α	hu	hz	hj
FCH	$h(s) = (h_{20}(s) + g(i)) \bmod n$	100%	16.40	21.94	12.44
hl	$h_j = (h(s) + j) \bmod m,\ \text{para } 1 \le j \le m-1$	65.195%	13.64	14.14	12.26
MWHC-2	$h(s) = g(h_1(s)) \oplus g(h_2(s))$	100%	14.88	20.22	10.62
hl	$h_j = (h(s) + j) \bmod m,\ \text{para } 1 \le j \le m-1$	40%	11.74	12.21	10.61
MWHC-3	$h(s) = g(h_1(s)) \oplus g(h_2(s)) \oplus g(h_3(s))$	100%	19.14	22.43	14.11
hl	$h_j = (h(s) + j) \bmod m,\ \text{para } 1 \le j \le m-1$	65.207%	13.66	14.15	13.45

Tabela 4.7: Comparação da utilização de FHPMs versus *hashing* linear considerando o tempo para pesquisar todas as URLs, variando as funções hu, hz e hj na composição das FHPMs e do *hashing* linear.

A Tabela 4.8 apresenta o *overhead* de espaço para cada FHPM considerada e para o *hashing* linear com fator de carga que o torna equivalente (em termos de eficiência) à FHPM logo acima considerando o conjunto de URLs. Comparando os resultados com os apresentados na Tabela 4.6, podemos observar que o método endereçamento aberto com tratamento de colisões por *hashing* linear se torna equivalente às FHPMs para fatores de carga um pouco maiores, diminuindo o *overhead* de espaço do método. No entanto, a função do algoritmo FCH ainda é mais eficiente em termos de espaço e tempo. Já as FHPMs geradas pelos algoritmos MWHC-2 e MWHC-3 serão mais eficientes em termos de espaço para $N_C \geq 2$ e $N_C \geq 3$, respectivamente.

Algoritmos	α	m	Tamanho do domínio da função g	*Overhead*	*Overhead* para $n = 10,935,928$
FCH	100%	n	$\left\lceil \frac{3.5n}{(\log n+1)} \right\rceil$	$N_B \times \left\lceil \frac{3.5n}{(\log n+1)} \right\rceil$	6.1MB
hl	65.195%	$\lceil 1.5339n \rceil$	0	$N_B \times \lceil N_C \times 0.5339n \rceil$	22.27MB
MWHC-2	100%	n	$2.09n$	$N_B \times \lceil 2.09n \rceil$	87.1MB
hl	40%	$\lceil 2.5n \rceil$	0	$N_B \times \lceil N_C \times 1.5n \rceil$	62.6MB
MWHC-3	100%	n	$1.23n$	$N_B \times \lceil 1.23n \rceil$	51.3MB
hl	65.207%	$\lceil 1.5336n \rceil$	0	$N_B \times \lceil N_C \times 0.5336n \rceil$	22.26MB

Tabela 4.8: Comparação da utilização de FHPMs versus *hashing* linear considerando o *overhead* de espaço para armazenar a tabela *hash* e o arranjo g de cada FHPM para a coleção de URLs.

Comparando os resultados obtidos para as duas coleções, podemos observar que o crescimento do tamanho médio das chaves no conjunto S faz com que as FHPMs fiquem

menos competitivas em relação ao *hashing* linear. Isso porque nas FHPMs sempre é preciso computar mais do que uma função *hash*, cuja complexidade depende do tamanho das chaves. Já no *hashing* linear somente uma função *hash* é computada. Mesmo assim as FHPMs se mostraram muito competitivas.

O ganho em espaço proporcionado pelas FHPMs é enorme em aplicações nas quais não há busca sem sucesso, pois não precisamos armazenar o conjunto de chaves, uma vez que colisões não ocorrem. Em muitas de tais aplicações o tamanho do conjunto de chaves não pode ser armazenado em memória interna. Isto torna o mecanismo de resolução de colisões muito lento, uma vez que muitos acessos aleatórios a disco devem ser efetuados. Como as FHPMs são livres de colisões, elas proporcionam também um enorme ganho de desempenho. Um exemplo de tais aplicações é o cálculo da *link* análise nas máquinas de busca, onde cada URL ocupa em média 60 caracteres e em geral existem milhões de URLs que participam do cálculo.

Para utilizar uma FHPM temos que gerá-la previamente para o conjunto de chaves. Devido a isso, um problema que ocorre é que o custo de inserção ou remoção de chaves em S tem a mesma complexidade do algoritmo para gerar a FHPM utilizada. Para os algoritmos da família MWHC-r [22] esta complexidade é $O(n)$. Nestas situações o método endereçamento aberto com tratamento de colisão por *hashing* linear é mais adequado. No entanto, para manter os custos de inserção e remoção constantes, temos que regerar a tabela *hash* toda vez que o fator de carga cresce ao ponto da taxa de colisão ultrapassar um certo limite previamente estabelecido de acordo com a eficiência desejada para o sistema de busca.

4.4 Geração da Função Hashing Perfeita Mínima

Nesta seção mostramos que é possível gerar FHPMs eficientemente e comparamos os algoritmos considerados. Como mostrado na Seção 4.3, o algoritmo FCH gera a FHPM que necessita de menos espaço para ser armazenada e o algoritmo MWHC-2 gera a FHPM mais eficiente. Assim, nesta seção vamos comparar os algoritmos em termos do tempo de geração das FHPMs.

4.4.1 Impacto das Funções *Hash* no Tempo de Geração das FHPMs

Nesta seção apresentamos a influência das funções hu, hz e hj no tempo de geração das FHPMs pelos algoritmos estudados. A Tabela 4.9 mostra os tempos de geração de FHPMs para o conjunto de chaves extraído da coleção TREC-VLC2 utilizando os algoritmos MWHC-2 e MWHC-3. O algoritmo MWHC-3 é o mais eficiente e mais estável (baixo desvio padrão), pois para $|V| = 1.23n$ a probabilidade de gerarmos um 3-grafo acíclico tende a 1 com o crescimento de n e isto faz com que o número médio de iterações para obter o 3-grafo acíclico seja próximo de 1.

O número de iterações esperadas para o algoritmo MWHC-2 é 2.92 (valor dado por $1/p$, onde p é calculado utilizando a Eq. (3.2)). A função hz foi a única que permitiu o valor de N_i ficar abaixo do esperado. Embora a função hu tenha as mesmas propriedades da função hz, o alto valor de N_i é devido a *overflows*, pois os valores utilizados no conjunto de pesos de hu estão no intervalo $[0, m-1]$. Como $m = n = 10,935,928$, as multiplicações e adições geraram valores muito grandes não sendo possível representá-los em 4 *bytes*. Este efeito não teve muitas conseqüências para o algoritmo MWHC-3, pois são utilizadas 3 funções *hash* ao invés de 2. Já a função hj causa mais iterações devido a sua alta taxa de colisões que provoca arestas múltiplas e ciclos no grafo.

	MWHC-2					MWHC-3				
Função	N_i	Map. e Ord.	Pes.	Total	DP	N_i	Map. e Ord.	Pes.	Total	DP
hz	2.5	42.3	3.3	45.6	31.6	1	23.4	4.2	27.6	0.09
hj	13.0	211.3	3.3	214.6	202.7	1.01	22.7	4.2	26.9	0.09
hu	7.5	126.6	3.3	129.9	127.7	1	23.9	4.2	28.1	0.09

Tabela 4.9: Tempo de geração de FHPMs para o conjunto de chaves extraído da coleção TREC-VLC2 utilizando os algoritmos MWHC-2 e MWHC-3, variando as funções hu, hz e hj na composição das FHPMs.

A Tabela 4.10 mostra os tempos de geração de FHPMs para o conjunto de chaves extraído da coleção TREC-VLC2 utilizando o algoritmo FCH. Ao contrário dos outros dois algoritmos, os passos de mapeamento e ordenação são estáveis e o passo de pesquisa é o que possui altas variações. Isso ocorre devido a necessidade de se obter uma função h_{20} perfeita para todos os *Buckets*. A dificuldade de obtermos a função h_{20} está relacionada com a distribuição dos tamanhos dos *Buckets* (vide Seção 4.4.2 para detalhes sobre a distribuição). A função hj gera *Buckets* maiores, por isso o maior valor para NIh_{20}, o

que implica numa menor eficiência de FCH. Uma outra coisa que influencia no tempo de geração das FHPMs é com quantos endereços vazios um padrão P_i é alinhado (N_{AL}), na média. Novamente, a função hj apresenta o maior valor e por esses dois motivos ela faz com que FCH seja mais lento para gerar uma FHPM. Já a função hu executa menos iterações para obter h_{20} mas como é mais lenta que hz e o seu valor de N_{AL} é maior, o algoritmo FCH fica mais lento utilizando hu do que utilizando hz.

Função	NIh_{20}	N_{AL}	Map.	Ord.	Pes.	Total	DP
hz	160.7	594.5	9.4	0.04	973.0	982.44	41.43
hj	1147.7	3851.5	9.0	0.04	5857.2	5866.24	541.00
hu	108.8	601.4	9.7	0.04	1029.3	1039.04	30.99

Tabela 4.10: Tempo de geração de FHPMs para o conjunto de chaves extraído da coleção TREC-VLC2 utilizando o algoritmo FCH.

Para avaliar o impacto do tamanho das chaves no desempenho dos algoritmos para gerar FHPMs realizamos os mesmos experimentos para a coleção de URLs. As Tabelas 4.11 e 4.12 apresentam os resultados obtidos para os algoritmos MWHC-2 e MWHC-3 e para o algoritmo FCH, respectivamente. Os resultados são semelhantes aos obtidos para a coleção TREC-VLC2, sendo que o tempo de geração é um pouco maior. Isso ocorre porque as chaves são maiores, o que diminui a eficiência do passo de mapeamento dos algoritmos, pois o custo de acesso à disco para ler o conjunto é maior e o custo para computar as funções *hash* aumenta com o tamanho das chaves. A variação que ocorre com a utilização da função hu no algoritmo MWHC-3 é devida ao aumento do tamanho das chaves. Como conseqüência os valores parciais gerados durante o cálculo de hu são maiores do que os gerados para as chaves da coleção TREC-VLC2. Assim, ocorreram mais *overflows* dificultando a geração do 3-grafo acíclico e aumentando o número de iterações.

	MWHC-2					MWHC-r				
Função	N_i	Map. e Ord.	Pes.	Total	DP	N_i	Map. e Ord.	Pes.	Total	DP
hz	3.2	101.0	3.3	104.3	69.3	1	39.6	4.2	43.8	0.11
hj	12.3	267.3	3.3	270.6	327.1	1	31.5	4.2	35.7	0.08
hu	4.3	112.5	3.3	115.8	73.1	1.5	52.7	4.2	56.9	24.3

Tabela 4.11: Tempo de geração de FHPMs para a coleção de URLs utilizando os algoritmos MWHC-2 e MWHC-3, variando as funções hu, hz e hj na composição das FHPMs.

Embora o algoritmo FCH gere a função que necessita de menos espaço para ser armazenada, ele é o algoritmo menos eficiente e como veremos na próxima seção é inviável utilizá-lo para grandes conjuntos de chaves, devido a sua complexidade exponencial. Já o

Função	NIh_{20}	N_{AL}	Map.	Ord.	Pes.	Total	DP
hz	168.7	595.8	16.6	0.04	2093.00	2109.64	139.35
hj	1470.1	3923.7	14.3	0.04	6098.36	6112.70	706.72
hu	227.0	596.7	16.0	0.04	1950.88	1966.92	178.65

Tabela 4.12: Tempo de geração de FHPMs para a coleção de URLs utilizando o algoritmo FCH.

algoritmo MWHC-3 é o mais eficiente. No entanto, a avaliação da função gerada é mais lenta. Já o algoritmo MWHC-2 é o segundo mais eficiente e gera as funções mais rápidas, embora necessitem de mais espaço para serem armazenadas.

4.4.2 Experimentos Sobre a Complexidade dos Algoritmos

Nesta seção apresentamos os experimentos para verificar a complexidade de tempo para gerar FHPMs utilizando cada um dos algoritmos considerados. Os gráficos apresentados nas Figuras 4.1 e 4.2 ilustram o comportamento do tempo total de execução dos algoritmos para as funções hu, hz e hj. As Tabelas 4.13, 4.14, 4.15, 4.16 e 4.17 contém os dados utilizados para gerar os gráficos e o detalhamento dos tempos de execuções em cada etapa dos algoritmos. Como o comportamento assintótico dos algoritmos é o mesmo independente do tamanho da chave, para esta seção foram realizados experimentos somente no conjunto de chaves extraído da coleção TREC-VLC2.

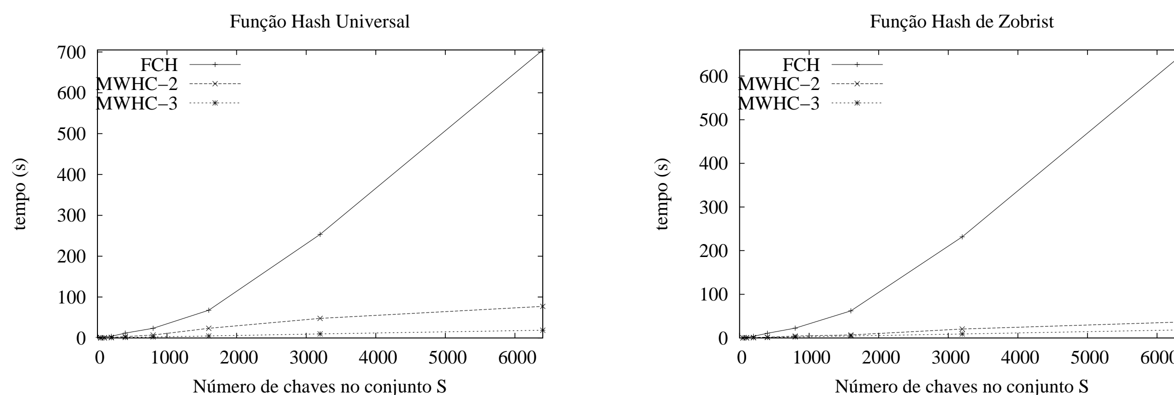

Figura 4.1: Tempo de execução dos algoritmos utilizando as funções hu e hz.

Observando os gráficos e as tabelas percebemos o comportamento linear dos algoritmos MWHC-2 e MWHC-3. Embora o comportamento do algoritmo FCH pareça quadrático, na verdade ele é exponencial, pois a medida que o número de chaves cresce, aumenta o número de *Buckets* de maior tamanho e isso faz com que o número de iterações para obter a função h_{20} tenda ao infinito.

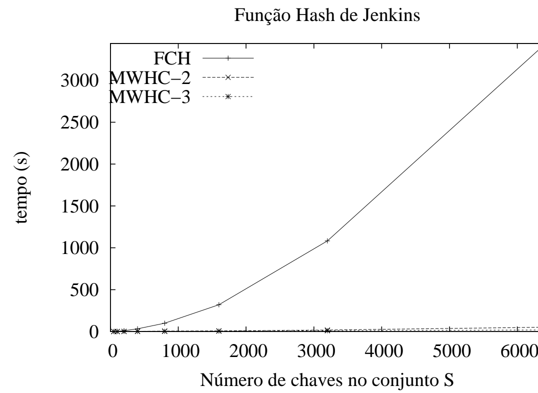

Figura 4.2: Tempo de execução dos algoritmos utilizando a função *hash* de Jenkins.

	MWHC-2 e função hu				MWHC-2 e função hz			
n	N_i	Map. e Ord.	Pes.	Total	N_i	Map. e Ord.	Pes.	Total
50,000	4.73	0.26	0.01	0.27	3.27	0.19	0.01	0.20
100,000	3.33	0.43	0.02	0.45	2.67	0.35	0.02	0.37
200,000	4.40	1.27	0.06	1.33	3.07	0.89	0.06	0.95
400,000	4.93	3.01	0.13	3.14	2.57	1.58	0.13	1.71
800,000	5.37	7.05	0.28	7.33	3.17	4.29	0.28	4.57
1,600,000	9.00	22.96	0.57	23.53	2.40	6.20	0.57	6.77
3,200,000	8.97	46.53	1.16	47.69	3.67	19.21	1.16	20.37
6,400,000	7.00	74.79	2.39	77.18	3.17	34.21	2.38	36.59

Tabela 4.13: Impacto das funções hu e hz no tempo de execução do algoritmo MWHC-2.

	MWHC-2 e função hj				MWHC-3 e função hu			
n	N_i	Map. e Ord.	Pes.	Total	N_i	Map. e Ord.	Pes.	Total
50,000	3.30	0.20	0.01	0.21	1.23	0.09	0.01	0.10
100,000	3.20	0.45	0.02	0.47	1.13	0.19	0.02	0.21
200,000	3.67	1.14	0.06	1.20	1.20	0.46	0.06	0.52
400,000	4.13	2.70	0.13	2.83	1.13	0.95	0.15	1.10
800,000	3.83	5.44	0.28	5.72	1.10	2.27	0.33	2.60
1,600,000	2.47	6.78	0.57	7.35	1.10	4.00	0.68	4.68
3,200,000	3.10	17.23	1.16	18.39	1.13	8.36	1.38	9.74
6,400,000	4.33	49.14	2.40	51.54	1.03	16.01	2.85	18.86

Tabela 4.14: Impacto das funções hj e hu no tempo de execução dos algoritmos MWHC-2 e MWHC-3, respectivamente.

	MWHC-3 e função hz				MWHC-3 e função hj			
n	N_i	Map. e Ord.	Pes.	Total	N_i	Map. e Ord.	Pes.	Total
50,000	1.00	0.08	0.01	0.09	1.10	0.08	0.01	0.09
100,000	1.00	0.18	0.02	0.20	1.30	0.19	0.02	0.21
200,000	1.00	0.40	0.06	0.46	1.17	0.41	0.07	0.48
400,000	1.00	0.86	0.15	1.01	1.24	0.90	0.15	1.05
800,000	1.00	2.13	0.33	2.46	1.31	2.20	0.33	2.53
1,600,000	1.00	3.71	0.68	4.38	1.26	3.85	0.68	4.53
3,200,000	1.00	7.60	1.38	8.98	1.33	7.85	1.39	9.24
6,400,000	1.00	15.68	2.85	18.53	1.35	16.19	2.89	19.08

Tabela 4.15: Impacto das funções hz e hj no tempo de execução do algoritmo MWHC-3.

	FCH e função hu					FCH e função hz				
n	NIh_{20}	Map.	Ord.	Pes.	Total	NIh_{20}	Map.	Ord.	Pes.	Total
50,000	20.83	0.03	0.0004	0.52	0.56	20.33	0.03	0.0004	0.44	0.47
100,000	20.00	0.07	0.0006	1.28	1.35	39.77	0.07	0.0005	1.11	1.17
200,000	17.60	0.14	0.0010	3.79	3.94	79.67	0.14	0.0011	3.47	3.60
400,000	21.57	0.29	0.0020	11.48	11.78	61.23	0.28	0.0020	10.39	10.67
800,000	27.10	0.61	0.0038	22.95	23.57	69.53	0.59	0.0037	21.76	22.35
1,600,000	50.27	1.28	0.0070	66.40	67.69	92.63	1.25	0.0071	60.76	62.02
3,200,000	71.03	2.64	0.0133	250.77	253.42	102.53	2.54	0.0133	229.02	231.57
6,400,000	68.93	5.47	0.0250	699.17	704.67	143.90	5.26	0.0252	649.04	654.33

Tabela 4.16: Impacto das funções hu e hz no tempo de execução do algoritmo FCH.

	FCH e função hj				
n	NIh_{20}	Map.	Ord.	Pes.	Total
50,000	123.20	0.03	0.0004	0.99	1.02
100,000	152.50	0.06	0.0007	2.85	2.92
200,000	192.53	0.13	0.0011	9.84	9.96
400,000	211.60	0.26	0.0019	32.18	32.44
8,00,000	343.00	0.55	0.0039	98.38	98.94
1,600,000	530.47	1.17	0.0070	319.16	320.33
3,200,000	525.53	2.41	0.0133	1080.27	1082.69
6,400,000	1065.70	5.02	0.0251	3421.15	3426.20

Tabela 4.17: Impacto da função hj no tempo de execução do algoritmo FCH.

A Figura 4.3 apresenta a distribuição dos tamanhos dos *Buckets* para $n = 1,600,000$, $n = 6,400,000$ e $n = 10,935,928$ quando utilizamos as funções hz e hj (não apresentamos o gráfico referente a função hu, pois possui o mesmo comportamento da função hz). Como podemos observar, a medida que o valor de n cresce há uma diminuição no número de *Buckets* de menor tamanho e um aumento do número de *Buckets* de maior tamanho. Além disso, a função hj tende a gerar *Buckets* maiores, pois provoca mais colisões. Essa é a razão do maior número de iterações para obter h_{20} quando utilizamos hj para representá-la.

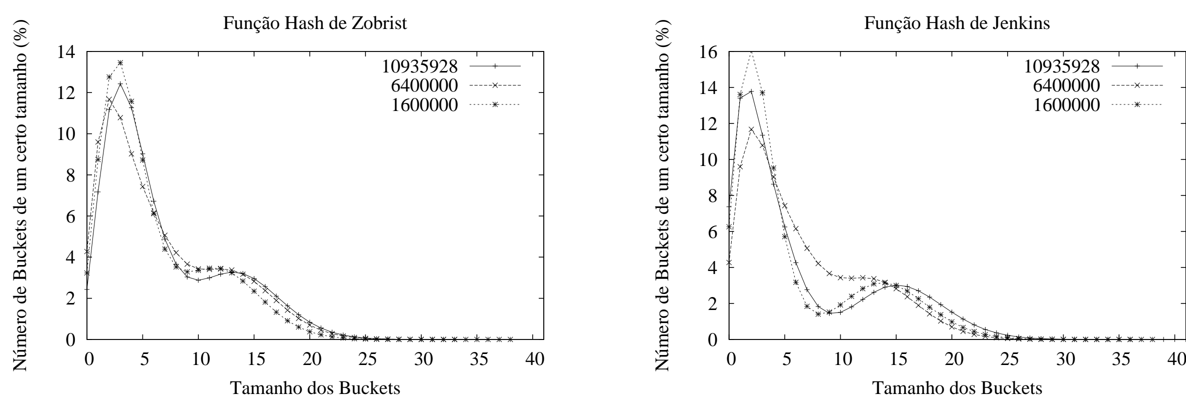

Figura 4.3: Distribuição do tamanho dos *Buckets* para as funções *hash* de Zobrist e Jenkins.

Por fim, através do gráfico apresentado na Figura 4.4, confirmamos o crescimento logarítmico do tamanho do maior *Bucket*. Embora as três funções tenham o mesmo comportamento, novamente verificamos que a função hj gera os maiores *Buckets*.

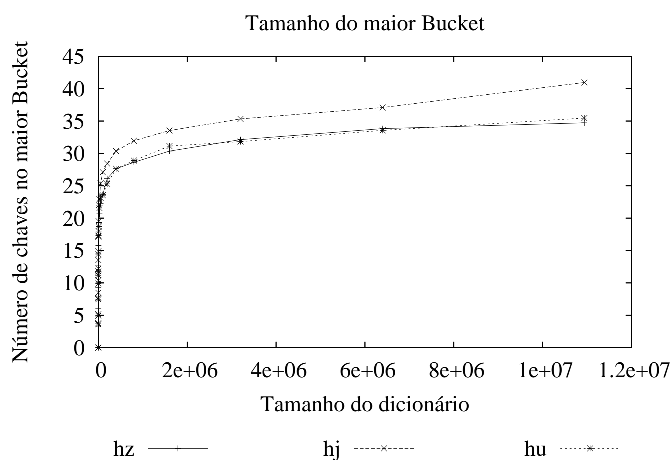

Figura 4.4: Tamanho do maior *Bucket*, $O(\log n)$.

Capítulo 5

Conclusões e Trabalhos Futuros

Nesta dissertação comparamos a utilização de funções *hash* perfeitas mínimas para endereçar uma tabela *hash* com o método endereçamento aberto, o qual resolve colisões por *hashing* linear. Mostramos que a utilização de FHPMs para endereçar a tabela é tão eficiente quanto o *hashing* linear com fator de carga acima de 35% e 40% para conjuntos de chaves onde os tamanhos médio das chaves são aproximadamente aproximadamente 8.52 e 57.23 caracteres, respectivamente. No entanto, a utilização de FHPMs é uma solução mais econômica em termos de espaço. Principalmente nos casos em que a busca sem sucesso não ocorre, pois para estas situações não precisamos armazenar o conjunto de chaves.

As FHPMs são apropriadas para conjuntos de chaves estáticos ou que sofrem alterações esporadicamente. No caso de conjuntos nos quais inserções ou remoções são freqüentes, é preferível a utilização do endereçamento aberto com *hashing* linear ou *hashing* perfeito dinâmico [8, Seção 7]. Em contrapartida, para conjuntos de chaves estáticos, principalmente os que contém uma grande quantidade de palavras chave, a utilização de *hashing* linear exige um maior *overhead* de espaço para obter o mesmo desempenho das FHPMs (vide Seção 4.3).

Uma consideração a ser feita é que nesta dissertação não levamos em conta alguns fatores que podem melhorar o desempenho do método endereçamento aberto que resolve colisões por *hashing* linear. Por exemplo, a popularidade de alguns termos nos conjuntos estáticos. No entanto, a principal vantagem das FHPMs em relação ao *hashing* linear é que elas viabilizam algumas aplicações nas quais não há busca sem sucesso e o conjunto de chaves não pode ser armazenado em memória interna. Para estas aplicações o *hashing* linear é muito ineficiente em termos de espaço e tempo.

Identificamos, descrevemos e comparamos os principais algoritmos da literatura para geração de FHPMs. Os algoritmos da família MWHC-r [22] são os mais eficientes e são capazes de gerar uma FHPM com complexidade ótima, $O(n)$. As FHPMs são de ordem preservada e por isso são armazenadas em $O(n \log n)$ *bits*. Já o algoritmo FCH [14] gera FHPMs que podem ser armazenadas em $O(n)$ *bits*, no entanto, o algoritmo tem complexidade exponencial. Mesmo tendo complexidade exponencial, conseguimos gerar FHPMs para um conjunto com $10, 935, 928$ palavras chave.

Por fim, estudamos o impacto de três funções *hash* não perfeitas no tempo para computar o endereço de uma certa palavra chave na tabela, tanto com a utilização de FHPMs quanto com a utilização de *hashing* linear para endereçar a tabela. A função *hash* proposta por Jenkins [20] foi a mais eficiente para compor as FHPMs. No entanto, devido a sua alta taxa de colisão, ela apresentou os piores resultados com o *hashing* linear aplicado à chaves de pequeno tamanho médio (aproximadamente 9 caracteres), onde é preferível a utilização da função proposta por Zobrist [31]. Já para chaves de maior tamanho médio (aproximadamente 57 caracteres), a função *hash* de Jenkins torna tanto as FHPMs quanto o *hashing* linear mais eficientes e a função de Zobrist faz com que os métodos sejam mais lentos (vide Seção 4.2).

O estudo realizado nesta dissertação permitiu a identificação de caminhos promissores de pesquisa a serem seguidos em trabalhos futuros. As principais oportunidades de continuidade de pesquisa a partir do trabalho aqui descrito são:

1. A propriedade de preservação de ordem não é importante para a maioria das aplicações em que métodos de *hashing* são utilizados. Assim, um caminho promissor de pesquisa é utilizar grafos cíclicos visando obter algoritmos mais eficientes para gerar e armazenar uma FHPM que não preserve a ordem. Um algoritmo ingênuo que trabalha com tais grafos é apresentado em [30, Seção 5.5.4]. No entanto, o algoritmo não consegue gerar uma FHPM para conjunto com mais de mil chaves.

2. A eficiência de avaliação das FHPMs geradas pelos algoritmos relacionados no Capítulo 3 pode ser melhorada. Além dos novos algoritmos trabalharem eficientemente sobre grafos cíclicos, um outro caminho promissor de pesquisa é melhorar a eficiência de avaliação das FHPMs geradas.

3. A geração de FHPMs para conjuntos de chaves onde o número de chaves ultrapassa 500 milhões e nos quais não há busca sem sucesso, não é possível com os algoritmos apresentados. Isso porque os métodos só foram propostos para memória primária.

Assim, um outro trabalho futuro interessante é pesquisar mecanismos para viabilizar a geração de FHPMs para grandes massas de dados utilizando memória secundária, já que não encontramos na literatura algoritmos capazes de gerar FHPMs neste contexto.

4. Por fim, um outro caminho para gerar FHPMs para grandes conjuntos de chaves é a utilização de compressão do r-grafo randômico. Recentemente, em [3], são apresentados alguns algoritmos para compressão de grafos. Para gerar os grafos e experimentar os algoritmos de compressão, os autores fazem uso de FHPMs para mapear um conjunto de URLs para os vértices de um grafo da *Web*.

Bibliografia

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, Reading, Massachusetts, 1999.

[2] J. Bentley. Programming pearls: A sample of brilliance. *Communication of The ACM*, (30):754–757, 1987.

[3] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *13th International World Wide Web Conference*, pages 595–602, 2004.

[4] C. C. Chang. The study of an ordered minimal perfect hashing scheme. *Communications of the ACM*, 27(4):384–387, December 1984.

[5] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, 1980.

[6] Z. J. Czech and B. S. Majewski. A linear time algorithm for finding minimal perfect hash functions. *The computer Journal*, 36(6):579–587, 1993.

[7] Z.J. Czech, G. Havas, and B.S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[8] Z.J. Czech, G. Havas, and B.S. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.

[9] J. Ebert. A versatile data structure for edges oriented graph algorithms. *Communication of The ACM*, (30):513–519, 1987.

[10] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, 1968.

[11] P. Flajolet, D. E. Knuth, and B. Pittel. The first cycles in an evolving graph. *Discrete Math*, 75:167–215, 1989.

[12] E. A. Fox, Q. F. Chen, A. M. Daoud, and L. S. Heath. Order preserving minimal perfect hash functions and information retrieval. *ACM Trans. Inform. Systems*, 9(3):281–308, July 1991.

[13] E.A. Fox, Q.F. Chen, A.M. Daoud, and L.S. Heath. Order preserving minimal perfect hash functions and information retrieval. In *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 279–311. ACM Press, 1990.

[14] E.A. Fox, Q.F. Chen, and L.S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 266–273, 1992.

[15] E.A. Fox, L. S. Heath, Q.Chen, and A.M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[16] E.A. Fox, L.S. Heath, and Q.F. Chen. An $o(n \log n)$ algorithm for finding minimal perfect hash functions. Technical report, Virginia Polytechnic Institute and State University, Blacksburg, VA, April 1989.

[17] G. Havas, B.S. Majewski, N.C. Wormald, and Z.J. Czech. Graphs, hypergraphs and hashing. In *19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*, number 790, pages 153–165, Utrecht the Netherlands, 1993. Springer Lecture Notes in Computer Science.

[18] D. Hawking. Overview of trec-7 very large collection track (draft for notebook), 1998.

[19] G. Jaeschke. Reciprocal hashing: A method for generating minimal perfect hashing functions. *Communications of the ACM*, 24(12):829–833, December 1981.

[20] Bob Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, 22(9), september 1997.

[21] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, third edition, 1998.

[22] B.S. Majewski, N.C. Wormald, G. Havas, and Z.J. Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.

[23] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, 1984.

[24] M. Molloy and B. Reed. The size of the giant component of a random graph with a given degree sequence. *Combinatorics, Probability and Computing*, 7(3):295–305, 1998.

[25] M. S. Neubert. Algoritmos distribuídos para a construção de arquivos invertidos. Master's thesis, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Março 2000.

[26] S. A. Özel and H. A. Güvenir. An algorithm for mining association rules using perfect hashing and database pruning. In *Tenth Turkish Symposium on Artificial Intelligence and Neural Networks (TAINN'2001)*, pages 257–264, June 2001.

[27] T. J. Sager. A polynomial time generator for minimal perfect hash functions. *Commun. ACM*, 28(5):523–532, May 1985.

[28] R. Sprugnoli. Perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM*, 20(11):841–850, November 1977.

[29] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Multimedia Information and Systems. Morgan Kaufmann, San Francisco, California, second edition edition, 1999.

[30] N. Ziviani. *Projeto de Algoritmos com implementações em Pascal e C.* Pioneira Thompson, segunda edição, 2004.

[31] A. L. Zobrist. A new hashing method with applications for game playing. *ICCA – International Computer-Chess Association journal*, 13(2):69–73, 1990.