Funções *Hashing* Perfeita e Mínima

CÉSAR FRANCISCO DE MOURA COUTO[1], DAVID MENOTI[1]

[1]Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Av. Antônio Carlos, 6627, Pampulha - Belo Horizonte, MG, CEP 31270-010, Brasil
{cesarfmc,menoti}@dcc.ufmg.br

Abstract. Neste artigo é apresentado um resumo sobre três abordagens de geração de Funções *Hashing* Perfeita e Mínima (FHPM). Uma delas mantém a ordem lexicográfica das chaves e as outras duas não. Duas dessas abordagens estão próximas dos limites inferiores de espaço de especificação para suas classes. Resultados analíticos e empíricos são apresentados incluindo o tempo de geração da FHPM para grandes conjuntos de chaves.

1 Introdução

Hashing (espalhamento) ou transformação de chave é um método de pesquisa onde os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa [Ziviani, 2004]. Esse método é constituído de duas fases:

- Calcular o valor da função de transformação (função *hashing*), a qual transforma a chave de pesquisa em um endereço na tabela.

- Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço da tabela, é necessário existir um método para lidar com colisões.

Uma função *hashing* deve mapear as n chaves em inteiros dentro do intervalo $\{0, \ldots, M-1\}$, no qual m é o tamanho da tabela. A função *hashing* ideal é aquela que seja simples de ser implementada e para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer. Quando colisões ocorrem algo deve ser feito para lidar com as mesmas. Uma das formas mais utilizadas para resolver colisões é a construção de uma lista encadeada para cada endereço da tabela *hash*. Dessa forma todas as chaves com o mesmo endereço na tabela são encadeadas na lista.

Existem algoritmos que geram funções *hashing* que evitam colisões, ou seja, não existem chaves que podem ser mapeadas para os mesmos endereços. Nessa classe estão as Funções de Hashing Perfeita e Hashing Perfeita Mínima. Uma função hashing transforma um conjunto de chaves x_j, $1 \leq j \leq n$, em um conjunto de valores inteiros no intervalo de $0 \leq h(x_j) \leq m-1$ com colisões permitidas. Quando não houver colisões, ou seja, quando $h(x_i) = h(x_j)$ se e somente se $i = j$, a função *hashing* é denominada função *hashing* perfeita [Ziviani, 2004].

Se o numero de chaves n e o tamanho da tabela m são iguais $\alpha = n/m = 1$ (fator de espalhamento), então a função *hashing* é denominada FHPM, ou seja, apenas um acesso a tabela é necessário para encontrar um registro e não há lugares vazios.

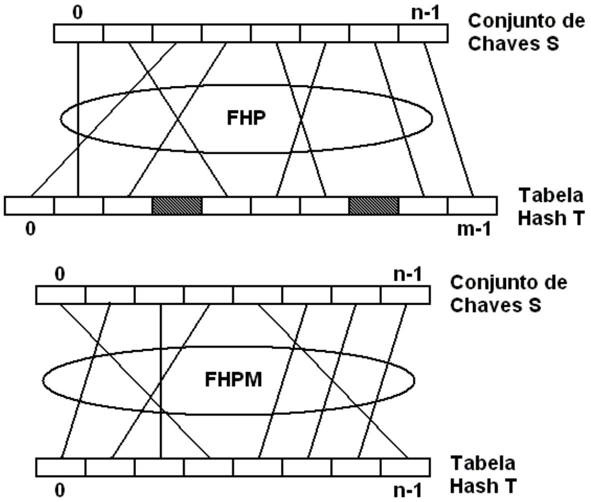

Figura 1: Função *Hash* Perfeita (FHP) e Função *Hash* Pefeita e Mínima (FHPM).

Neste artigo serão apresentados um resumo sobre três abordagens. Uma delas versa sobre um algoritmo que gera a FHPM e mantém a ordem lexicográfica das chaves [Czech et al., 1992]. As outras duas apresentam algoritmos que também geram a FHPM mas não mantêm a ordem lexicográfica das chaves [Fox et al., 1992a], [Fox et al., 1992b]. Todas as três abordagens estão próximas do limite inferior de espaço de especificação para as FHPM de suas classes [Mehlhorn., 1982],[Mehlhorn, 1984] e os autores clamam que os algoritmos são lineares em tempo esperado.

O artigo é organizado da seguinte forma. Na Seção 2 são apresentados três algoritmos para a construção da FHPM. Os experimentos realizados são apresentados na

Seção 3. As conclusões são apontadas na Seção 4.

2 Algoritmos para geração de Função *Hash* Perfeita e Mínima

Para facilitar a discussão, define-se a seguinte terminologia:

- U: universo das chaves. $|U| = N$.
- S: conjunto de chaves. $S \subset U, |S| = n \ll N$.
- T: tabela *hash*. $|T| = m, m \geq n$.
- h: função *hash*. $h : U \rightarrow T$.
- h é uma FHP: sem colisão, h é de um para um em S.
- h é uma FHPM: sem colisões (i.e. FHP) e $m = n$.

Para um dado conjunto de chaves S tomado do universo U, deseja-se obter uma FHPM que mapeie qualquer chave k em S para uma única posição na tabela *hash* T.

Os dois primeiros algoritmos (Algoritmo 1 e Algoritmo 2) que serão apresentados nessa seção são referentes a classe de FHPM que não preservam a ordem e essa classe tem um limite inferior de espaço de especificação de aproximadamente $1,44$ *bits* por chave. O terceiro algoritmo (Algoritmo 3) a ser apresentado pertence a classe de FHPM que preserva a ordem lexicográfica das chaves e essa classe tem um limite inferior de espaço de especificação de $\log_2 n$ *bits* por chave, que é atingido pelo Algoritmo 3.

No entanto, antes de apresentar os algoritmos, apresenta-se a abordagem de MOS (*Mapping*, *Ordering* e *Searching*) para contextualizá-los.

2.1 MOS - Abordagem de Cicheli e Sager

Existem muitas estratégias para encontrar Funções *Hashing* Perfeitas. A estratégia que será descrita utilizará a abordagem MOS (*Mapping*, *Ordering* e *Searching*) definada por Cichelli [Cichelli, 1980].

A abordagem é uma estratégia dividida em três passos. O primeiro é o passo de *Mapping* que consiste em transformar um conjunto de chaves de um universo original para um novo universo. O segundo é o passo de *Ordering* que coloca as chaves em uma seqüência ordenada que determina a ordem em que os valores das chaves serão espalhados (*Hashing*). O passo de *Ordering* deve particionar as chaves dentro de subseqüências de chaves consecutivas. Uma subseqüência é um nível e as chaves de cada nível devem ter seus valores determinados na tabela *hash* ao mesmo tempo. O terceiro passo é o de *Searching* que consiste em tentar determinar valores *hash* para as chaves de cada nível. Se o passo de *Searching* encontrar chaves de um nível onde ele é incapaz de acomodá-las, ele deve voltar para um nível anterior, determinar novos valores para as chaves do nível e tentar novamente determinar valores para os níveis posteriores. O método de Sager é um bom exemplo para a abordagem MOS.

Sager [Sager, 1984],[Sager, 1985] propôs a formalização e extensão da aproximação de Cichelli [Cichelli, 1980]. Como Chichelli, ele assumiu que uma chave é uma cadeia de caracteres. Três funções auxiliares *hash* são definidas para mapear as chaves em um universo U.

$$
\begin{aligned}
h_0 &: U \rightarrow \{0, \ldots, m-1\} \\
h_1 &: U \rightarrow \{0, \ldots, r-1\} \\
h_2 &: U \rightarrow \{0, \ldots, 2r-1\}
\end{aligned}
$$

onde r é um parâmetro tipicamente igual a $m/2$ que determina quanto de espaço o passo de *Mapping* gasta para armazenar as funções *hash* perfeitas. Essas funções auxiliares comprimem cada chave k dentro de uma identificador único, que é uma tripla de inteiros em um novo universo de tamanho mr^2. A classe de funções pesquisadas onde g é uma função na qual os valores são selecionados durante a pesquisa.

Através do passo de *Mapping* as chaves foram transformadas para triplas e de triplas para um grafo bipartido (grafo de dependência) no qual os vértices são $h_1()$ e $h_2()$ e as arestas representam as chaves. As duas partes do grafo bipartido são os conjunto de vértices $(0, \ldots, r-1)$ e $(r, \ldots, 2r-1)$. Para cada chave k, existe uma aresta rotulado por k conectando $h_1(k)$ e $h_2(k)$.

No passo de *Ordering* Sager empregou uma heurística baseada em encontrar pequenos ciclos no grafo, chamada *mincycle*. Em cada iteração do passo de *Ordering* a heurística encontra um conjunto de arestas que pertence a pequenos ciclos. Este conjunto de chaves correspondem a um conjunto de arestas que correspondem ao próximo nível na ordenação. Não existe nenhuma prova que a heurística *mincycle* encontra a Função *Hashing* Perfeita Mínima (FHPM), mas na prática essa heurística é bem sucedida para um conjunto de algumas centenas de chaves. *Mincycle* gasta $O(m^4)$ de tempo e $O(m^3)$ de espaço, enquanto que no passo subsequente (*Searching*) geralmente é gasto somente $O(m)$. Assim o tempo e o espaço requerido na execução da heurística *mincycle* é a principal barreira para grandes conjuntos de chaves.

Uma análise de esquemas para *Hashing* Perfeito mostra que a maioria desses esquemas são geralmente aplicáveis para pequenos conjuntos de chaves, ou requerem uma quantia de espaço e tempo muito grande. A mais competitiva estratégia encontrada vem do trabalho de Fredman, Komlós e Szemérdi [FredMan et al., 1984] no qual é utilizado um universo que é um conjunto de inteiros $\{1, \ldots, n\}$ em constraste com o que será proposto nas abordagens, onde chaves são arbitrariamente cadeias de caracteres.

2.2 Algoritmo 1

Após uma cuidadosa análise do algoritmo de Sager descrito acima, três observações cruciais foram feitas para servir de base para construção desse algoritmo, conforme descrito em [Fox et al., 1992b].

- Mostra-se que deve-se explorar a aleatoriedade sempre que possível. Deve-se negociar uma pequena probabilidade de falha para um excelente caso médio de desempenho.
- A distribuição do grau dos vértices é altamente assimétrica. Isto pode ser explorado para realizar o passo de *Ordering* de uma maneira mais eficiente.
- Determinar valores de g para um conjunto de palavras pode ser visto como tentar ajustar um padrão dentro de um disco parcialmente preenchido, onde é importante entrar com grandes padrões enquanto o disco está parcialmente cheio.

Estas três observações permitem obter o passo de *Ordering* em tempo linear. Portanto, desde que o passo de *Mapping* e *Seaching* tenham tempo esperado $O(n)$, a complexidade de tempo desse algoritmo é $O(n)$.

2.2.1 *Mapping*

No passo de *Mapping* um conjunto de triplas é obtido para servir de identificador da chave. A técnica para construir esse conjunto de triplas consiste essencialmente em obter um número aleatório (mod n) para cada chave, fazendo o uso de todas as informações na chave para dar o máximo de discriminação. Os números aleatórios são obtidos por geradores de números aleatórios e estão entre 0 e $2^{32} - 2$. Considere o uso desses números aleatórios para obter as triplas desejadas. Primeiro, três tabelas de números aleatórios criadas, uma para cada função h_0, h_1 e h_2, onde cada tabela contém um número aleatório para cada possível caractere de cada posição i na chave. Então as triplas são computadas usando as seguintes fórmulas:

$$\begin{aligned} h_0(k) &= (\textstyle\sum_{i=1}^{y} table_{0i}(k_i))\text{mod } n \\ h_1(k) &= (\textstyle\sum_{i=1}^{y} table_{1i}(k_i))\text{mod } r \\ h_2(k) &= (\textstyle\sum_{i=1}^{y} table_{2i}(k_i))\text{mod } r + r \end{aligned}$$

Essas triplas quase sempre são distintas umas das outras, no entanto, quando ocorre, casualmente, alguma tripla igual, o processo é repetido até que todas as triplas sejam distintas.

2.2.2 *Ordering*

A distribuição do grau dos vértices no grafo de dependência é decididamente assimétrica e realmente tem um número interessante de propriedades. Isso mostra que em um grafo de dependência, a maioria dos vértices têm grau pequeno e podendo ser explorado para obter níveis pequenos no passo de *Ordering*. A distribuição assimétrica dos graus dos vértices fornece a inspiração para a criação de uma nova heurística, chamada de heurística de *Ordering*. Em vez de ordenar arestas (chaves) de um grafo de dependência, como na heurística *mincycle*, descrita anteriormente, a nova heurística ordena os vértices. Seja $v_1, v_2, \ldots, v_{2r}$ alguma ordenação de um conjunto de arestas $K(v_i)$ que vai de v_i para vértices anteriores na ordem

$$K(v_i) = \{(v_i, v_j) \in E \mid j < i\}$$

Este conjunto de arestas representa um conjunto de chaves e toda chave ocorre exatamente em um $K(v_i)$. $K(v_i)$ deve ser vazio mas não pode ser maior do que os graus de v_i. A ordenação do conjunto de chaves dentro de níveis é exatamente a ordenação dos $K(v_i)$'s não vazios.

2.2.3 *Searching*

Para cada iteração do passo *Searching*, um nível da ordenação é colocado na tabela *hash*. Cada nível contém um conjunto de chaves $K(v_i)$ correspondentes ao vértice v_i. Cada chave k pertencente a $K(v_i)$ (conjunto de chaves de um nível) tem o mesmo valor h_2, logo $h_2(k) = v_i$, consequentemente ter-se-ia o mesmo valor $g(h_2)$, logo $g(h_2(k)) = g(v_i)$. Por suposição o valor $g \circ h_1$ de k já está selecionado. Desde que todos os valores h_0 já estejam determinados, $h(k)$ é determinado pela escolha do valor de $g(v_i)$. Considere a soma de dois valores já definidos de k. Seja

$$b(k) = h_0(k) + g(h_1(k))$$

Então

$$h(k) = (b(k) + g(v_i))(\text{mod } n)$$

Os valores de $b(k)$ para todos as chaves k pertencentes a $K(v_i)$ são transformados de $g(v_i)(\text{mod } n)$ para os valores *hash* das chaves.

A tabela *hash* é vista como um disco com n posições, onde essas posições são preenchidas pelo passo de *Searching*. Para determinar os valores da tabela *hash* para as chaves em $K(V_i)$, o passo de *Searching* deve determinar o valor de $g(v_i)$ e colocar todos os valores de $(b(k) + g(v_i))(\text{mod } n)$ em posições vazias da tabela *hash*. Assim ao final desse passo a tabela *hash* é construída.

2.3 Algoritmo 2

O algoritmo para geração de Funções *Hash* Mínima e Perfeita discutido nessa seção é extraído do artigo [Fox et al., 1992a]. Nesse artigo foi desenvolvio um novo algoritmo que elimina a necessidade de várias multiplicações, produzindo a FHPM com espaço de especificação abaxo de 2.5 bits por chave e é relativamente fácil de entender e implementar. Este algoritmo que será apresentado resumidamente aqui é descrito com maior riqueza de detalhes em [Chen, 1992].

O Algoritmo 2 corrige muitos problemas existentes no Algoritmo 1. Primeiro, o Algoritmo 1 faz uso de tabelas de tamanho moderadamente grandes para especificar o mapeamento de caracteres que mascaram as chaves. Nesse algoritmo, através do uso e extensão do método de Pearson [Pearson, 1990] (função *Hash* universal [Ziviani, 2004],[Cormen et al., 2002]), tabelas de mapeamento usando somente 128 caracteres são produzidas. Segundo, no Algoritmo 1, a fase de *Searching* é menos sofisticada, exigindo muito operação sem sucesso para localizar soluções aceitáveis. Nesse algoritmo, através da adição de uma estrutura de dados auxiliar, o tempo de *Searching* foi reduzido significativamente. Por último, o Algoritmo 2 reduz o espaço de especificação da FHPM mudando radicalmente as fases de *Mapping*, *Ordering* e *Searching*, em relação ao Algoritmo 1. Mais precisamente, nenhum uso de grafo bipartido com dependência é feito como sugerido por Sager [Sager, 1985]. Em vez disso, S é relacionado a s em dois passos:

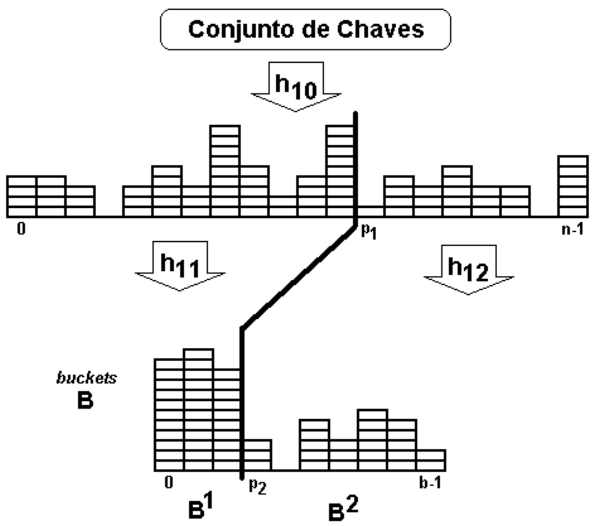

Figura 2: Passo de *Mapping* do Algoritmo 2.

- As chaves são mapeadas para um conjunto B de *buckets* (ver Figura 2)
- As chaves em cada *bucket* são separadamente mapeadas em T (ver Figura 3)

Objetivando-se ter um espaço medido em *bits* por chaves ao invés de palavras por chaves, é necessário procurar por valores cujos números sejam proporcionais à $\lceil cn/(\log_2 n + 1)\rceil$ ao invés de n, como feito no Algoritmo 1. Isto parcialmente explica a necessidade de introduzir os *buckets* no processo.

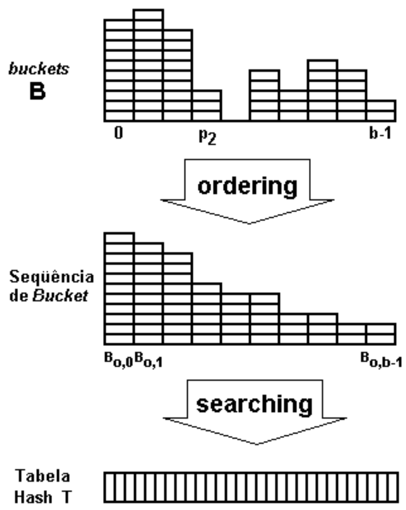

Figura 3: Passo de *Ordering* e de *Searching* do Algoritmo 2.

Os passos de *Mapping* e de *Ordering* e *Searching* são ilustrados nas Figuras 2 e 3, respectivamente. Mais detalhes são dados nas subseções seguintes.

2.3.1 *Mapping*

O passo de *Mapping* cumpre com vários objetivos importantes. Primeiro, cada um das n chaves são mapeadas em um valor inteiro, na faixa $0, \ldots, n-1$. Isto é feito por uma função *hash* pseudo-aleatória h_{10} que mapeia várias chaves em alguns valores e pode deixar alguns endereços sem nenhuma chave. Observe uma ilustração deste processo no topo da Figura 2 .

$$h_{10} : S \rightarrow \{0, \ldots, n-1\}$$

Segundo, deseja-se encolher a faixa de valores inteiros de n para b de forma que mais tarde seja preciso procurar somente entre b valores. Encontrar uma FHPM que tenha espaço de especificação perto do limite inferior pode ser realizado quando c é próximo de 2, i.e., quando b é aproximadamente $2n/\log_2 n$. Pode-se realizar isto através da composição de h_{10} em outra função que irá mapear as chaves em uma faixa $0, \ldots, b-1$.

Entretanto, neste processo pode-se realizar um terceiro objetivo. Em particular, deseja-se separar as chaves em dois grupos maiores. A segunda função é, de fato, modelada por duas funções h_{11} e h_{12} que operam sobre duas porções disjuntas de $0, \ldots, n-1$.

$$h_{11} : \{0, \ldots, p_1 - 1\} \rightarrow \{0, \ldots, p_2 - 1\}$$
$$h_{12} : \{p_1, \ldots, n - 1\} \rightarrow \{p_2, \ldots, b - 1\}$$

Estas funções, junto com h_{10}, realizam o mapeamento das chaves para os *buckets*.

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

Assim, a função de mapeamento $bucket(k)$ é composta de três funções: h_{10} que distribui as chaves aleatoriamente em um conjunto auxiliar $\{0, \ldots, n-1\}$, h_{11} e h_{12} que, por sua vez, distribuem aleatoriamente esses valores em B, em particular, em dois subconjuntos B^1 e B^2 de tamanhos desiguais. Observe que h_{11} e h_{12} dependem de dois parâmetros p_1 e p_2. Bons valores para estes dois parâmetros são determinados experimentalmente com $p_1 = 0,6n$ e $p_2 = 0,3n$.

Isto significa que aproximadamente 60% das chaves (pois $p_1 = 0,6n$ e h_{10} é provável de ter uma distribuição relativamente uniforme em uma resolução baixa) são mapeadas em aproximadamente 30% dos *buckets* (pois $p_2 = 0,3n$), i.e. $B^1 = \{0, \ldots, p_2 - 1\}$. De fato, força-se que cada um dos *buckets* produzidos por h_{11} possuam muitas chaves. O trabalho anterior [Fox et al., 1992b] deixa indícios que grandes grupos de chaves podem ser gerenciados caso eles sejam manipulados primeiramente no processo de busca (*Searching*).

Ao mesmo tempo, os outros 40% de chaves são "espalhados" por h_{12} em 70% dos *buckets*, i.e. $B^2 = \{p_2, \ldots, b-1\}$, produzindo menos chaves por *buckets*. É desejável ter pequenos grupos de chaves a serem processados no final do processo de *Searching*.

Em resumo, o passo de *Mapping* ilustrado na Figura 2, realiza os objetivos de mapear os inteiros, encolher as faixas de chaves de inteiros, e separar as chaves em pequenos e grandes grupos.

2.3.2 Ordering

Durante o passo de *Ordering*, ilustrado no topo da Figura 3, usa-se a organização desenvolvida durante o passo de *Mapping* para preparar o passo de *Searching*. Os *Buckets* são ordenados pelo tamanho (número de chaves) em ordem descrescente para obter a seqüência de *buckets*:

$$\{B_{o,0}, B_{o,1}, \ldots, B_{o,b-1}\}$$

Análises indicam que devido ao uso de funções randômicas no passo de *Mapping*, pode-se estimar o número de *buckets* de cada tamanho. Mesmo para conjunto de chaves muito grandes os maiores buckets terão tamanhos relativamente pequenos. Uma única passada sobre os *buckets* irá produzir a seqüência de *buckets* desejada. O passo de *Searching* irá processar todas as chaves em um *bucket*, e o processamento será realizado dos *buckets* maiores para os menores.

2.3.3 Searching

O passo de *Searching* envolve a escolha de um parâmetro $g()$ para cada um dos *buckets*, de forma que cada chave em um *bucket* possa ser mapeada através da função *hash* h finalmente construída, para uma posição previamente não usada na tabela *hash* T.

Essencialmente, o grupo de chaves em um *bucket* deve ser "ajustado" em T ao mesmo tempo, pois eles são mutuamente restringidos devido ao processo anterior que os colocaram em um mesmo *bucket*. A escolha do valor parâmetro $g()$ para o *bucket* deve garantir que seus "padrões" de entrada possam ser "ajustados" em uma posição não ocupada em T. Como diferentes valores de $g()$ são testados, os padrões são "rotacionados" até que um bom casamento/ajuste seja encontrado.

O passo de *Searching* mapeia chaves em cada bucket $B_{o,i}$ para T através da função h:

$$h_{20} : S \times 0,1 \rightarrow \{0, \ldots, n-1\}$$
$$h(k) = \{h_{20}(k, d) + g(B_{o,1})\} \text{mod } n.$$

Esta função *hash* h final tem uma forma simples e é fácilmente computável para qualquer chave em S. Ela é formada pela soma de dois valores.

- h_{20} é uma função pseudo-aleatória mapeando chaves em cada $B_{o,i}$ para valores distintos em $\{0, n-1\}$. Lembrando que $\log_2 n + 1$ *bits* são alocados para cada *bucket*. Um bit designado d entre esses bits é usado por h_{20} como parte da semente. Como 0 e 1 podem ser os valores para d, h_{20} pode gerar dois conjunto de inteiros para as chaves em $B_{o,i}$. Isto adiciona um grau de liberdade para o passo de *Searching*, evitando falhas através da troca de valores d quando necessário. Um inteiro r global a todos os *buckets* também pode ser usado como parte da semente para h_{20}. Algum *bucket* pode falhar ao ser mapeado para inteiros distintos, um novo valor para r pode ser tentado, rotacionando todos os *buckets* já processados. Com a ajuda de r, a mesma seqüência de *bucket* pode ser mantida.

- Cada $g(B_{o,i})$ toma $\log_2 n$ *bits*

- $g(B_{o,i})$ rotaciona o conjunto de padrões (as chaves em um *bucket* $B_{o,i}$) para um ajuste.

2.4 Algoritmo 3

As principais referencias para o algoritmo descrito nessa seção são [Czech et al., 1992], [Czech et al., 1997]. Este algoritmo é baseado em grafos aleatórios para encontrar a FHPM e tem função *hash* da forma:

$$h(k) = (g(h_1(k)) + g(h_2(k))) \mathrm{mod}\, n$$

onde h_1 e h_2 são funções que mapeiam as chaves compostas de caracteres em inteiros, e g é uma função que mapeia inteiros na faixa $\{0, \ldots, n-1\}$. O autor clama que o algoritmo tem complexidade de tempo esperada $O(n)$. O espaço exigido para armazenar a FHPM é $O(n \log_2 m)$ *bits*, que é o ótimo para FHPM com ordem preservada (ver [Mehlhorn., 1982],[Mehlhorn, 1984] e [Havas and Majewski, 1992])

Este algoritmo para geração da FHPM pode ser visto como o seguinte problema. Para um dado grafo não direcionado $G = (V, E)$, $|E| = n$, $|V| = m$ encontrar uma função $g : V \rightarrow \{0, \ldots, m-1\}$ de forma que a função $h : E \rightarrow \{0, \ldots, m-1\}$ definida como:

$$h(e = (u, v) \in E) = (g(u) + g(v)) \mathrm{mod}\, n$$

é uma função bijetora. Em outras palavras procura-se por uma atribuição de valores aos vértices de forma que cada aresta é a soma dos valores associados aos vértices que a determinam tomado o módulo do número de arestas é um único inteiro na faixa {0,n-1}. A geração do grafo é detalhada na Seção 2.4.1.

Este problema não apresenta solução sempre se um grafo arbitrário é considerado. Entretanto, se um grafo é acíclico, um procedimento muito simples pode ser usado para encontrar valores para cada vértice que será apresentado na Seção 2.4.2.

É possível estabelecer uma analogia entre a abordagem MOS [Cichelli, 1980], [Sager, 1984], [Sager, 1985] utilizada pelos Algoritmos 1 e 2 com a implementada por este problema que possui basicamente dois passos: *Mapping* e *Assignment*. Para tanto é necessário que a fase de geração do grafo seja entendida como sendo uma fase de *Mapping*, pois está havendo um mapeamento das chaves para um outro espaço o grafo. A fase de *Ordering* pode ser visto como sendo a imposição de que o grafo gerado seja acíclico, pois isto agiliza o processo de *Searching* que poderá ser realizado em um única busca em um grafo ($O(|E| + |A|)$). E a fase de *Searching* é desempenhada em uma única tentativa dado que o grafo seja acíclico.

2.4.1 *Mapping* e *Ordering*

Conforme dito anteriormente, para a geração da FHPM é necessário que um grafo acíclico seja gerado. Este grafo $G = (V, E)$ é gerado através de duas funções *hash* auxiliares h_1 e h_2, onde $V = \{0, \ldots, m, -1\}$, com m determinado posteriomente, e $E = \{h_1(k), h_2(k) : k \in S\}$. As funções auxiliares h_1 e h_2 são funções pseudo-aleatórias que mapeiam S em $\{0, \ldots, n-1\}$. Existem várias forma de determinar essas funções aleatórias conforme descrito em [Czech et al., 1992].

Em [Czech et al., 1992] é mostrado que o número esperado de iterações para a geração de um grafo acícilo pode ser considerado constante dado o número m de vértices e n arestas, onde $m = cn$. Detalhes probabilísticos podem ser vistos em [Majeswki et al., 1992]. Brevemente pode-se dizer que a probabilidade de se ter um grafo acíclico (p_a) tende a $exp(-\sum_{k=1}^{m} 2^k/(2kc^k)$. Aplicando-se o limite ao exponecial quando $n \rightarrow \infty$ tem-se que: para $c > 2$ $p_a = \sqrt{(c-2)/c}$ e para $c \leq 2$ $p_a = 0$. Assim, para $c > 2$ a probabilidade de gerar um grafo acíclico aproxima-se de uma constante maior que zero.

2.4.2 *Searching*

Nesse passo, para este algoritmo, não existem tentativas de encontrar uma conjunto de valores que satisfaçam uma condição. Aqui é garantido que o grafo será rotulado e portanto a FHPM será gerada, então essa passado pode ser chamado de atribuição ou *Assignment*.

Então a criação do grafo pode ser definida da seguinte forma. Associe com cada aresta um único número $h(e) \in \{0, n-1\}$ em qualuqer ordem. Para cada componente conectado em g escolha um vértice v. Para este vértice, ajuste $g(v)$ para 0. Percorra o grafo usando uma busca em largura (ou qualquer outra busca regular em um grafo), começando pelo vértice v. Se qualquer vértice w é alcancável por u e o valor associado com a aresta $e = (u, w)$ é $h(e)$, ajuste $g(w)$ para $(h(e) - g(u)) \mathrm{mod}\, n$. Aplique o método acima para cada componente de G.

3 Experimentos

Foram realizados experimentos com uma base de mais de 3 milhões de chaves, a base de palavras chaves do TodoBR. Desse conjunto foram escolhidos subconjuntos de tamanho variados: 25000, 50000, 100000, 200000, 400000 e 800000 chaves. Os três algoritmos implementados foram testados com esses subconjuntos. Cada subconjunto foi executado seis vezes para obter-se um tempo médio de execução. O resultado desses experimentos são apresentados nas Figuras 4, 5 e 6 e uma comparação com o tempo de execução dos algoritmos é apresentado na Figura 7.

Nas Figuras 4, 5 e 6 são mostrados os tempos de cada passo dos três algoritmos implementados mais o tempo total de execução. Para os Algoritmos 1 e 2 são apresentados em separados os tempos dos passos de *Mapping*, *Ordering* e *Searching*, nas Figuras 4 e 5, respectivamente. Já na

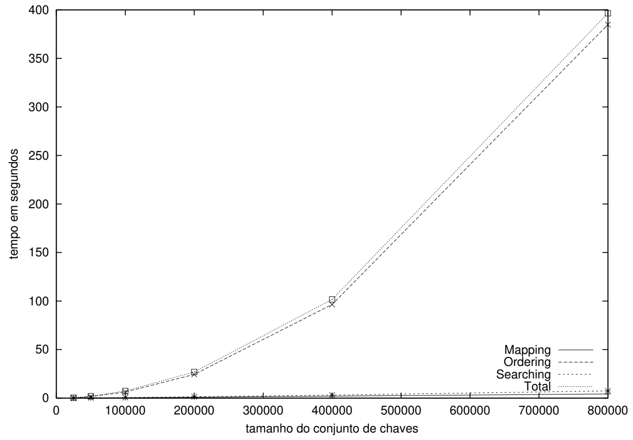

Figura 4: Resultados de Tempo para a implementação do Algoritmo 1 segundo [Fox et al., 1992b].

Figura 6 são apresentados os tempos dos passos de *Mapping* e *Assignment* para a implementação do Algoritmo 3. Lembrando que, caso queira ser feita uma analogia do Algoritmo 3 com a abordagem MOS e os Algoritmos 1 e 2, deve-se relacionar o passo de *Mapping* do Algoritmo 3 com os passos de *Mapping* e *Ordering* dos Algoritmos 1 e 2 e o passo de *Assignment* do Algoritmo 3 com o passo de *Searching* dos Algoritmos 1 e 2.

Observando-se os resultados de tempo apresentados na Figura 4 para o Algoritmo 1 ([Fox et al., 1992b]), conclui-se que a implementação feita tem ordem de complexidade quadrática, ou seja $O(n^2)$. E o passo responsável por essa grandeza é o passo de *Ordering*, já que os passos de *Mapping* e *Searching* são lineares, ou seja $O(n)$. Fica difícil perceber essa linearidade através do gráfico, mas isto é um fato que poderia ser confirmado pela tabela dos resultados, que não é exposta por razões de espaço do artigo.

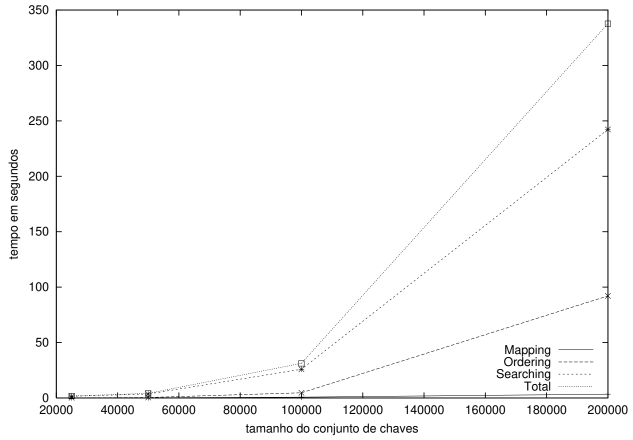

Figura 5: Resultados de Tempo para a implementação do Algoritmo 2 segundo [Fox et al., 1992a].

Através de uma análise dos resultados de tempo da implementação do Algoritmo 2 ([Fox et al., 1992a]) apresentados na Figura 5, pode-se concluir que a implementação feita do Algoritmo tem ordem de complexidade quadrática, ou seja $O(n^2)$. E além do comportamento quadrático verificado pelo passo de *Ordering*, conforme já visto no Algoritmo 1, o passo de *Searching* também apresenta esse comportamento. E o mesmo pode-se dizer, em relação ao Algoritmo 1, do passo de *Mapping*, que ele é linear. Esse comportamento quadrático não era esperado por ambos algoritmos (1 e 2) conforme descrito em seus artigos originais ([Fox et al., 1992b],[Fox et al., 1992a]). Em [Czech et al., 1997] é apresentado uma crítica a ordem de complexidade de ambos algoritmos, provando, matematicamente que eles são exponenciais. Todavia, a implementação feita pelos autores aqui, apresenta ordem de complexidade quadrática e os autores dos artigos originais apresentam resultados empíricos mostrando a linearidade dos algoritmos. Deve-se então verificar com mais cautela os detalhes de implementação dos artigos originais para confirmar as suposições.

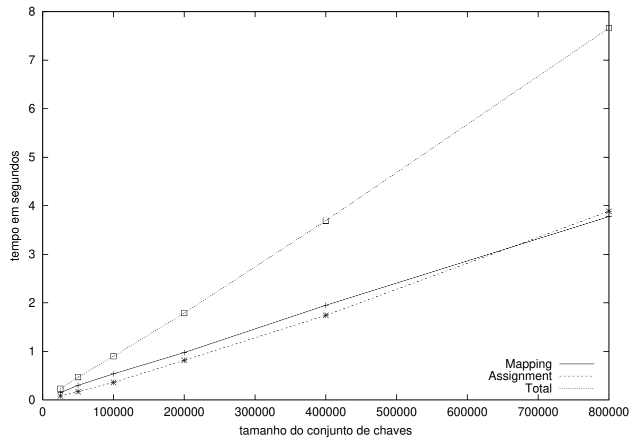

Figura 6: Resultados de Tempo para a implementação do Algoritmo 3 segundo [Czech et al., 1992].

Os resultados de tempo apresentados na Figura 6 relativos a implementação do Algoritmo 3 [Czech et al., 1992] confirmam a ordem de complexidade linear, $O(n)$, da implementação feita do Algoritmo 3. E, os tempo dos passos de *Mapping* e *Assignment* são quase idênticos.

Uma comparação entre os tempos de execução da implementação dos três algoritmos descrito no artigo é apresentada na Figura 7. Uma análise do gráfico dessa figura permite dizer que o Algoritmo 3 é o mais rápido, seguido de longe pelo Algoritmo 1. Como pode ser visto na Figura 7 o Algoritmo 2 é o mais lento de todos e não foi conseguido resultados em tempo hábil para conjuntos de chaves de 400000 e 800000.

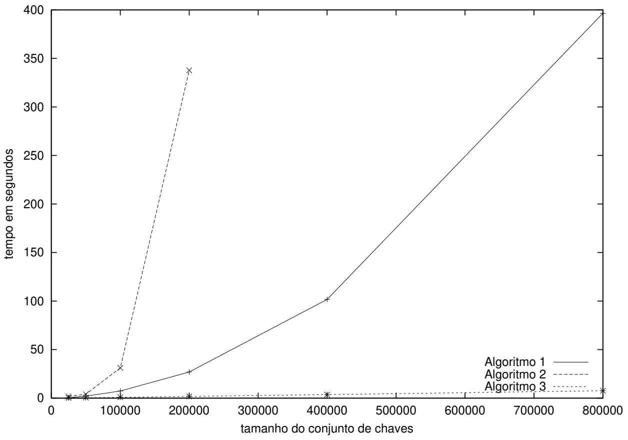

Figura 7: Uma comparação de tempo de execução da implementação dos três ([Fox et al., 1992b], [Fox et al., 1992a], [Czech et al., 1992]) algoritmos descritos no artigo.

4 Conclusões

Neste artigo foram apresentados três algoritmos para a geração da FHPM. Um deles (Algoritmo 3) mostrou-se muito rápido para grandes massas de dados, em comparação com os outros dois algoritmos, preservando a ordem lexicográfica das chaves, tendo espaço de especificação da FHPM $n \log_2 n$ *bits* sendo ótimo para essa classe de geração de FHPM. Mas $m = cn$ e $c > 2$. Os outros dois algoritmos (Algoritmo 1 e Algoritmo 2) pertencem a outra classe de problema, não preservam a ordem lexicográfica das chaves, tem espaço de especificação da FHPM (menos de $5n$ *bits*) superior ao limite inferior da classe. No entanto, seus espaços de especificação da FHPM são melhores que o do Algoritmo 1.

Referências

[Chen, 1992] Chen, Q. F. (1992). *An object-oriented database system for efficient information retrieval applications*. PhD thesis, Department of Computer Science, Virginia Tech.

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

[Cormen et al., 2002] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2002). *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition edition.

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

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

[Fox et al., 1992a] Fox, E. A., Chen, Q. F., and Heath, L. S. (1992a). 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.

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

[FredMan et al., 1984] FredMan, M. L., Komlós, J., and Szemerédi, E. (1984). Storing a sparse table with $o(1)$ worst-case access time. *Journal of ACM*, 31(3):538–544.

[Havas and Majewski, 1992] Havas, G. and Majewski, B. S. (1992). Optimal algorithms for minimal perfect hashing. Technical Report 234, The University of Queensland,Key Centre for Software Technology.

[Majeswki et al., 1992] Majeswki, B., N.C. Wormald, Z. C., and Havas, G. (1992). A family of generators of minimal perfect hash functions. Technical Report 16, DIMACS, Rutgers University, New Jersey, USA.

[Mehlhorn., 1982] Mehlhorn., K. (1982). On the program size of perfect and universal hash functions. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science (FOCS'82)*, pages 170–175.

[Mehlhorn, 1984] Mehlhorn, K. (1984). *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, Berlin - Heidelberg - New York - Tokyo.

[Pearson, 1990] Pearson, P. K. (1990). Fast hashing of variable-length text strings. *Communications of the ACM*, 28(6):667–680.

[Sager, 1984] Sager, T. J. (1984). A new method for generating minimal perfect hashing functions. Technical report, Department of Computer Science, University of Missouri-Rolla, Mo.

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

[Ziviani, 2004] Ziviani, N. (2004). *Projeto de Algoritmos com implementações em Pascal e C*. Pioneira Thomson Learnig, São Paulo - SP - Brasil, 2a. edition. revisada e ampliada.