UFES – Universidade Federal do Espírito Santo
PPGI – Programa de Pós-Graduação em Informática
Projeto e Análise de Algoritmos – 2009/1 – Mestrado em Informática
Prof. Claudine Santos Badue Gonçalves

Resumo: Funções Hash

Eleu Lima Natalli, Leonardo Araújo Peruch

{eleuln, theleoap}@gmail.com

1. Introdução

Na Ciência da Computação, uma tabela dispersão (também conhecida por tabela de espalhamento, tabelas esparsas ou ainda tabela *hash*, do inglês *hash*) é uma estrutura de dados especial, que associa chaves de pesquisa a valores. A utilização de tabelas *hash* em estrutura de dados é algo importante e com aplicações em diversas áreas (utilizadas para implementar vetores associativos, conjuntos e *caches*, indexação de grandes volumes de informação, etc), visto que fornecem um acesso muito mais rápido aos elementos de um vetor, por exemplo, que qualquer outro método (listas encadeadas, árvores binárias, etc). Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado.

A implementação típica busca uma função de dispersão (função *hash*) que seja de complexidade O(1), não importando o número de registros na tabela (desconsiderando colisões). O ganho com relação a outras estruturas associativas (como um vetor simples) passa a ser maior conforme a quantidade de dados aumenta. A função *hash* é a responsável por gerar um índice a partir de determinada chave. Caso a função seja mal escolhida, toda a tabela terá um mau desempenho. O ideal para a função *hash* é que sejam sempre fornecidos índices únicos para as chaves de entrada. A função perfeita (*hashing* perfeito) seria a que, para quaisquer entradas A e B, sendo A diferente de B, fornecesse saídas diferentes [Wikipedia].

Este trabalho é focalizado em cima da tese de mestrado do Fabiano [BOTELHO, 2004], a qual realiza um estudo vasto de forma comparativa acerca de *hashing*. Apresentaremos o estudo sobre algumas funções *hash* e os resultados experimentais mostrados nos testes.

2. *Hashing*

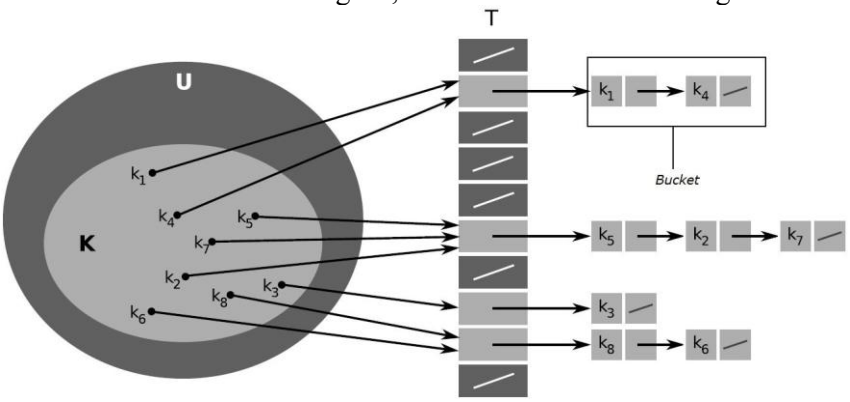
Uma **função *hash*** h(x) transforma os elementos de um universo arbitrário em inteiros positivos relativamente pequenos que indexem uma tabela - **a tabela de *hash*** - generalizando-se assim a operação de indexação, ou seja, os registros armazenados na tabela *hash* são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.

Como descrito em Ziviane [ZIVIANI, 2004], existem dois problemas relacionados com o método *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**. O domínio das chaves de uma tabela *hash* é tipicamente muito maior do que o número de entradas da tabela. É inevitável que duas chaves diferentes acabem sendo mapeadas para a mesma entrada da tabela pela função de dispersão.

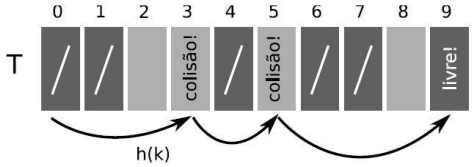
Uma forma de lidar com o problema das colisões é usar **endereçamento aberto**. Nesta abordagem, todos os registros são armazenados na própria tabela. Colisões são resolvidas através da localização do próximo espaço livre na tabela após o endereço fornecido pela função

de dispersão. Esta procura é feita de forma circular na tabela. Outra possibilidade de lidar com colisões é **encadear** as entradas da tabela cujas chaves resultaram no mesmo endereço calculado pela função de dispersão (gerando *Buckets*, ilustrado abaixo). Desta forma, cada endereço da tabela corresponde ao início de uma lista encadeada (*Bucket*) de registros.

Em uma tabela de *hash* encadeada, todos os elementos mapeados para uma mesma posição são armazenados em uma lista ligada, conforme ilustrado na imagem abaixo.



Em uma tabela de *hash* com endereçamento aberto, todos os elementos são armazenados na tabela propriamente dita. O espaço gasto com encadeamento é economizado e a colisão é tratada com a busca de uma nova posição para inserção, conforme a ilustração do na imagem abaixo.



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** (FHP). 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** (FHPM). Uma FHPM pode evitar totalmente o problema de desperdício de espaço e de tempo [BOTELHO, 2004].

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 frequentes em técnicas de mineração de dados [BOTELHO, 2004].

3. Funções *Hash*

Segundo Knuth [KNUTH, 1998], 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, foram selecionadas três funções *hash* para serem avaliados no trabalho do Fabiano [BOTELHO, 2004].

3.1 Função *Hash* Universal

Função *hash* universal (*hu*) onde a probabilidade de ocorrência de colisão é 1/*m*, sendo *m* o número de entradas da tabela *hash*. Sendo o 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 -> M é definida por:

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

A figura abaixo apresenta o programa utilizado para os pesos de uma certa função de *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)));
}
```

A função *hash* universal é muito simples de ser implementada, como ilustra na figura abaixo. Segundo Knuth [KNUTH, 1998], 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));
}
```

A probabilidade de ocorrência de colisão ha função *hu* é 1/*m*. Para provar, Fabiano [BOTELHO, 2004] utilizou a prova de que a família de funções *HU* é realmente uma família universal.

A complexidade da *hu* é O(*T*), sendo *T* o tamanho da chave. A complexidade para a função que gera os pesos é O(t_{max}). Vale observar que a função para gerar os pesos é utilizada somente na criação da tabela *hash*.

3.2 Função *Hash* Zobrist

Função *hash* de Zobrist (*hz*) é computada através do número de adições da função *hash* universal, mas com nenhuma multiplicação efetuada. Isto faz com que a *hz* seja computada mais eficientemente do que a função *hash* universal. Seja um conjunto de pesos P = { $P_{i,j}$ | $0 \leq i < t_{max}$ e $0 \leq j < |\sum|$ }, sendo cada $P_{i,j}$ um número inteiro escolhido aleatoriamente dentro do intervalo M = [0,*m*-1], a função *hz*: U -> M é definida por:

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

A figura abaixo apresenta o programa utilizado para os pesos de uma certa função de *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)));
}
```

A função *hash* de Zobrist é ainda mais simples de ser implementada, como ilustra a figura abaixo.

```
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);
}
```

A complexidade da *hz* é igual a *hu*, ou seja, O(T), sendo T o tamanho da chave. A complexidade para a função que gera os pesos é O(t_{max} x *TamAlfabeto*). Embora nenhuma multiplicação seja efetuada, a quantidade de memória interna para armazenar *hz* é O(t_{max} x *TamAlfabeto*) enquanto que para a função *hash* universal é O(t_{max}).

3.3 Função *Hash* Jenkins

A função *hash* de Jenkins (*hj*) foi projetada para atender os seguintes requisitos [BOTELHO, 2004]: (i) permitir que as chaves sejam arranjos de caracteres de tamanho variável. O tamanho dos arranjos deve variar de 8 a 200 *bytes*; (ii) ser mais eficiente que as funções existentes; (iii) permitir qualquer tamanho de tabela, inclusive potência de 2; (vi) distribuir uniformemente as chaves nas entradas da tabela.

O algoritmo, através de operações com *bits*, manipula os bytes da chave para calcular o valor da função *hash*. Segue abaixo o algoritmo macro da *hj* onde 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*. 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);
```

A complexidade da *hz* é igual a *hu*, ou seja, O(T), sendo T o tamanho da chave.

4. Resultados Experimentais

Essa seção que mostra os resultados experimentais que o Fabiano [BOTELHO, 2004] apresentou em sua tese de mestrado.

4.1 Dados utilizados nos experimentos

O Fabiano [BOTELHO, 2004] em sua tese utilizou o conjunto S de chaves nos experimentos, que é o vocabulário extraído da coleção TREC-VLC2 (*Very Large Collection 2*) e um conjunto de URLs coletadas da web:

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

Trata-se de dois conjuntos com iguais quantidades de elementos, diferindo apenas no tamanho das chaves. O conjunto de URLs possui o tamanho médio das chaves maior.

Durante a análise experimental dos algoritmos, foi calculado o fator de carga (α) que é dado pela fórmula: $\alpha = n/m$, onde n é o número de chaves e 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. Consequentemente, o tempo para se pesquisar uma chave S é degradado com o aumento do fator de carga.

4.2 Testes utilizando funções *hash* com método endereçamento aberto

Para comprovar que com o crescimento do fator de carga (consequentemente maior a possibilidade de acontecer colisões) o tempo de pesquisa para um chave em S é degradado, foi realizado o seguinte experimento:

- Para valores de α, foi gerada uma tabela *hash* com todas as chaves da dada coleção e aferiu-se os números médios de colisões por chave;
- Após isso, foi medido o tempo para pesquisar todas as chaves do conjunto S na tabela, utilizando as funções *hu*, *hz* e *hj*. As chaves foram pesquisadas na ordem em que elas aparecem no conjunto S.

A tabela abaixo mostra fatores de carga entre 70% e 25%, aferidos da coleção TREC-VLC2. As funções *hu* e *hz* apresentam aproximadamente a mesma taxa de colisão. Porém *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. A maior desvantagem da *hj* é que ela necessita de um fator de carga menor para ter o mesmo desempenho das *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

E a tabela abaixo retrata o mesmo teste agora utilizando o conjunto de URLs coletadas da web. Mesmo apresentando uma maior taxa de colisão, com o aumento do tamanho médio das chaves e fatores de carga abaixo de 65%, a *hj* é mais eficiente do que as *hz* e *hu*, pois executa 60% e 30% das instruções executas por elas. O conjunto de pesos da *hz* é mais custoso (gerando latência do clico da CPU - *cache misses*[1]) diminuindo a eficiência da mesma.

[1] Um *cache miss* refere-se a uma tentativa fracassada de ler ou escrever uma peça de dados no *cache*, o que resulta em um acesso à memória principal com muito mais tempo latência.

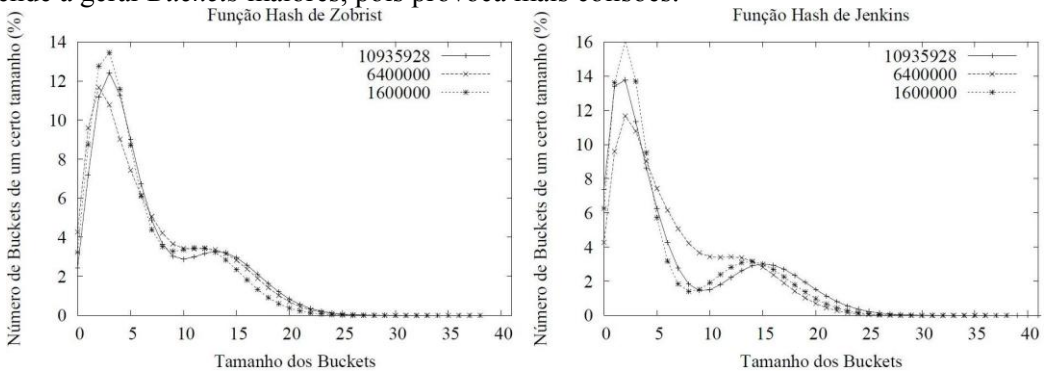
	hu		*hz*		*hj*	
α	Colisões/n	Tempo (s)	Colisões/n	Tempo (s)	Colisões/n	Tempo (s)
70%	1.17	14.37	1.16	14.87	381.69	85434.55
65%	0.93	13.63	0.93	14.14	0.94	12.00
60%	0.75	13.06	0.75	13.56	0.88	11.84
55%	0.61	12.62	0.61	13.10	0.81	11.65
50%	0.50	12.28	0.50	12.76	0.73	11.34
45%	0.41	11.98	0.41	12.43	0.62	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

Independente da função *hash* utilizada, pode-se observar que a taxa de colisão decresce com a diminuição do fator de carga (aumento do tamanho da tabela *hash*).

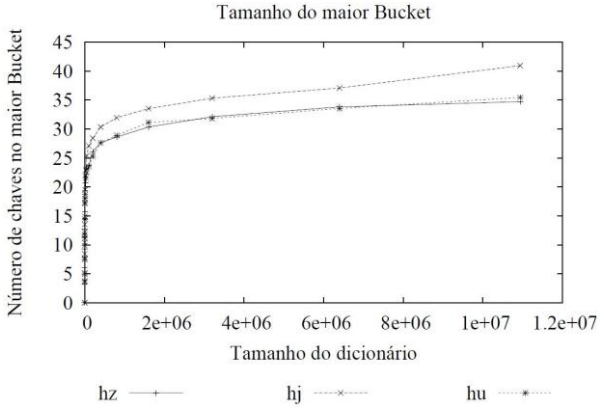
4.3 Testes dos algoritmos com método de encadeamento

Aqui é apresentado os experimentos que o Fabiano [BOTELHO, 2004] utilizou em sua tese para verificar crescimento do *Bucket* de cada um dos algoritmos das funções *hashs* estudadas. Quando as colisões ocorriam, aqui, as chaves são concatenadas na posição (encadeamento descrito na seção 2).

O gráfico abaixo apresenta a distribuição dos tamanhos dos *Buckets* para n = 1600000, n = 6400000 e n = 10935928 quanto é utilizado as funções *hz* e *hj* (não foi apresentado o gráfico referente a *hu* pois esta apresenta o mesmo comportamento da função *hz*). Como pode-se observar que a medida que o valor de n cresce há uma diminuição no número de *Bucktes* 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.



O gráfico abaixo confirma o crescimento logarítmico do tamanho do maior *Bucket*. Embora as três funções tenham o mesmo comportamento, novamente é verificado que a função *hj* gera maiores *Buckets*.



5. Considerações Finais

Estudamos três funções *hash* não perfeitas e, através dos resultados experimentais, Fabiano [BOTELHO, 2004] identificou que a função proposta por Jenkins foi a mais eficiente.

5.1 Conclusão

- *hj* apresenta maior taxa de colisão;
- *hj* necessita de um fator de carga menor para ter o mesmo desempenho que *hu* e *hz* (maior desperdício de espaço);
- Com o aumento do tamanho das chaves *hj* é mais eficiente;
- A medida que o número de chaves cresce *hj* tende a gerar mais colisões;
- Tamanhos das chaves afetam o desempenho das funções *hash*;
- Com o aumento to tamanho médio das chaves a *hj* é mais eficiente;
- O conjunto de pesos da *hz* é mais custoso (gerando *cache misses*), afetando seu desempenho;
- De modo geral a taxa de colisão decresce com a diminuição do fator de carga;

5.2 Trabalhos Futuros

Estudo acerta de *hashing* perfeito pode apresentar melhores resultados (principalmente em termos de espaço - principalmente nos casos em que a busca sem sucesso não ocorre, pois para estas situações não é preciso armazenar o conjunto de chaves).

6. Referências

BOTELHO, F. C.; Estudo Comparativo do Uso de Hashing Perfeito Mínimo. Dissertação (Mestrado em Informática). Universidade Federal de Minas Gerais, 2004.

BOTELHO, F.C., ZIVIANI, N.; Near-Optimal Space Perfect Hashing Algorithms. In Proceedings of the 22nd SBC Theses and Dissertations Contest (CTD'09), Bento Gonçalves, Brasil, Julho 2009.

KNUTH, D. E.; The Art of computer Programming: Sorting and Searching, volume 3 Addison-Wesley, third edition, 1998.

ZIVIANI, N.; Projeto de Algoritmos com Implementações em Pascal e C. Pioneira Thompson, segunda edição, 2004.

Wikipedia; Endereço http://pt.wikipedia.org/wiki/Tabela_de_dispers%C3%A3o . Acessado em Julho de 2009.