Capítulo 7

“Hash” universal e perfeito

Neste capítulo vamos estudar alguns aspectos importantes sobre os métodos de “hash” e, em especial o “hash” universal e “hash” perfeito. Os métodos de “hash” são de certo modo uma generalização dos algoritmos baseados na indexação (ver a Secção 5.1.2, página 80) permitindo tempo médio $O(1)$ nas operações de dicionário[1]. Mais especificamente, pretende-se construir uma função (de “hash”) que transforme 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. Com o método tradicional de “hash” consegue-se obter tempos de pesquisa, inserção e eliminação constantes (o que é excelente), mas apenas em termos do tempo médio – não no pior caso.

Pré-requesitos. Pressupomos que o leitor tem já um conhecimento genérico dos métodos de “hash” e da sua eficiência; neste capítulo vamos focar a nossa atenção em alguns tópicos mais avançados.

7.1 Considerações gerais sobre os métodos de “hash”

Nesta Secção vamos rever alguns dos conceitos ligados aos métodos de “hash”.

7.1.1 Universos grandes, funções de “hash”

Os dicionários são uma das estruturas básicas utilizadas em programação e a sua implementação eficiente é de importância fundamental. Podemos considerar 2 tipos de dicionários:

– Dicionários estáticos: dado um conjunto N de elementos, pretendemos representá-los numa

[1] Neste Capítulo supomos que a função de “hash” é calculada em tempo constante.

estrutura de informação que permita uma procura eficiente. São aplicações possíveis o dicionário das palavras reservadas de uma determinada linguagem de programação e o dicionário que contém as palavras correctamente escritas de uma lingua (por exemplo, do Português).

Uma implementação razoável é um vector ordenado dos elementos; a eficiência da pesquisa binária é $O(\log n)$. O "hash" permite uma implementação ainda mais rápida[2], em termos de tempo médio e mesmo (ver 7.3) no pior caso!

– <u>Dicionários dinâmicos</u>: as operações possíveis são a pesquisa, a inserção e, possivelmente, a eliminação.

  O vector ordenado é uma má implementação dos dicionários dinâmicos uma vez que a inserção é uma operação ineficiente, $O(n)$. São alternativas melhores: as árvores B e as árvores balanceadas. O método de "hash" é ainda melhor, em termos do tempo médio, uma vez que permite tempos constantes para as 3 operações.

Entre as muitas aplicações do "hashing", referimos a implementação eficiente dos métodos de tabelação ("memoization", ver página 124); estes métodos são usados, por exemplo, em pesquisas heurísticas, e em jogos – por exemplo, para guardar o valor de configurações de um tabuleiro de xadrês.

Formalmente temos

– Um universo U de cardinalidade $u = |U|$. Pode ser, por exemplo, o conjunto de todas as sequências de caracteres com comprimento 100 ($u = 127^{100}$).

– Um conjunto $N \subseteq U$ de elementos a representar na tabela de "hash". A cardinalidade de N é $n = |N|$. O conjunto N poderia ser constituído, por exemplo, pelos nomes dos alunos da Faculdade de Ciências.

– Uma tabela (vector) A de "hash" de cardinalidade $a = |A|$. Normalmente $a \ll u$, tendo a e n valores não muito diferentes, por exemplo, $a = 2n$.

– Define-se uma função de "hash" $h : U \to A$ que a cada valor de U atribui uma posição na tabela A. Para inserir x, coloca-se x na posição $h(x)$ da tabela A, *se essa posição estiver vazia*.

– Quando se insere um valor x e posteriormente um valor y com $h(y) = h(x)$, temos uma *colisão*. A posição da tabela para onde y deveria ir já está ocupada e terá que existir um método para resolver as colisões.

[2] De ordem $O(1)$ se a tabela de "hash" estiver bem dimensionada.

Resolução das colisões

Existem diversos métodos de resolver colisões que se podem classificar globalmente em 2 classes

a) *Métodos internos:* quando há uma colisão o valor a inserir é colocado na própria tabela A no próximo índice livre, sendo a noção de "próximo" dependente do método.

a) *Métodos externos:* cada posição da tabela A contém um apontador para uma lista ligada, inicialmente vazia. Cada valor x a inserir é colocado como primeiro elemento da lista. Neste caso, o haver ou não colisão é irrelevante; a não existência de colisão significa simplesmente que a lista está vazia.

Por hipótese usaremos um método externo.

Propriedades desejáveis da função de "hash"

O que se pretende é que, sendo a não muito maior que n, haja em média poucas colisões. Pretendemos que o método de "hash" tenha as seguintes propriedades:

a) *Espalhamento:* os índices $h(x)$ deverão ficar uniformemente espalhados na tabela, de modo a que haja poucas colisões.

b) *Tabela pequena:* para uma constante pequena k é $a \leq ks$; por outras palavras, a tabela não deverá ser muito maior que o número de elementos a inserir.

c) *Computação rápida:* para todo o elemento x, $h(x)$ deve ser computável em tempo $O(1)$.

Exemplo. Para o caso de um dicionário dos (digamos) 10 000 alunos da Faculdade, poderíamos escolher $a = 20\,000$ e a seguinte função de "hash"

$$((c_1 \ll 12) + (c_2 \ll 8) + (c_{n-1} \ll 4) + c_n) \mod 20\,000$$

onde $n \ll m$ significa que a representação binária de n é deslocada para a esquerda de m bits (o valor correspondente é multiplicado por 2^m).

Infelizmente um simples argumento de contagem, mostra-nos que as propriedades enunciadas podem ser difíceis de obter.

Teorema 21 *Para qualquer função de "hash" h existe um conjunto de $\lceil u/a \rceil$ valores que são mapeados num mesmo índice da tabela A.*

Exercício 79 *Demonstre o Teorema anterior.*

Para o exemplo dado atrás, concluimos que, independentemente da função de “hash” escolhida, existe um conjunto de elementos do universo (isto é, de “strings” de comprimento 100) com cardinalidade $\lceil 127^{200}/20\,000 \rceil > 10^{400}$ que é mapeado num único índice da tabela.

Felizmento veremos que o conhecimento prévio dos elementos a incluir na tabela permite (com um algoritmo aleatorizado) determinar uma função h e uma dimensão da tabela a não muito maior que s tais que não exista qualquer colisão! Este é um assunto que trataremos nas secções 7.2 e 7.3.

Eficiência do método clássico de “hash”

O método de “hash” tradicional (interno e externo) tem sido exaustivamente analisado. Em particular temos no caso médio e supondo que o cálculo de $h(x)$ é efectuado em tempo constante

– Pesquisa, inserção e eliminação: ordem de grandeza $O(1)$. A constante multiplicativa correspondente depende essencialmente do factor $\alpha = n/a$ que mede o grau de preenchimento da tabela.

No pior caso, todas aquelas operações têm tempo de ordem de grandeza $O(n)$, ver o Teorema 21. Ver (muitos) mais pormenores, por exemplo em [6].

7.1.2 Variantes do método de “hash”

Em linhas gerais, temos 3 métodos de “hash”:

1) **“Hash” clássico**: a função de “hash” está fixa e os dados têm uma determinada distribuição probabilística. consegue-se

tempo médio $O(1)$

Contudo, no pior caso, (se tivermos muito azar com os dados) os tempos das operações básicas (pesquisa, inserção e eliminação) são de ordem $O(n)$. Supõe-se que o leitor está a par desta análise; breve referência em 7.1.1

2) **“Hash” universal**, aleatorização proveniente da escolha da função de “hash”. Consegue-se

tempo médio $O(1)$, *quaisquer que sejamos dados*

Ver 7.2.

3) **“Hash” perfeito**, os valores a memorizar são conhecidos previamente[3]: Consegue-se determinar h por forma que

tempo $O(1)$ mesmo no pior caso

Ver 7.3.

No fundo, estamos a aleatorizar o algoritmo do "hash", da mesma forma que aleatorizamos o algoritmo clássico do "quick sort", ver página 63.

7.2 "Hash" universal: aleatorização do "hash"

Vimos que, qualquer que seja a função de "hash" escolhida, existem conjuntos N de dados que são maus no sentido em que para todo o $x \in N$, $h(x)$ é constante (com uma excepção, todas as listas ligadas de H ficam vazias), ver o Teorema 21. Suponhamos agora que a função de "hash" é escolhida aleatóriamente de um conjunto H de funções; para conjuntos e distribuições probabilísticas convenientes podemos conseguir que, qualquer que seja o conjunto de dados de entrada – não há maus conjuntos de dados –, o valor esperado do tempo de pesquisa seja $O(1)$; estamos obviamente a falar de "valor esperado" relativamente à distribuição probabilística das funções de "hash".

Como já dissemos, o "hash" universal é algo de muito semelhante ao "quick sort" aleatorizado onde, qualquer que seja o vector a ordenar – não há maus vectores – o tempo médio de execução[4] é de ordem $O(n \log n)$.

Definição 11 *Seja H um conjunto de funções de "hash" de U em $\{1, 2, \ldots, a\}$ associada a uma distribuição probabilística a que chamamos também H. Dizemos que H é* universal *se para todo o par de valores $x \neq y$ de U temos*

$$\text{prob}_{\,h \in_U H}[h(x) = h(y)] \leq 1/a$$

onde $h \in_U H$ significa que a função h é escolhida do conjunto H de forma uniforme, isto é, com igual probabilidade.

Por outras palavras, a probabilidade de haver uma colisão entre 2 quaisquer elementos distintos não excede $1/a$ (praticamente o valor mínimo) onde, lembra-se, a é o tamanho da tabela de "hash".

O seguinte resultado mostra que, relativamente a uma escolha uniforme $h \in_U H$, se esperam poucas colisões entre um qualquer valor x e os outros elementos de um qualquer conjunto N (existente na tabela de "hash"). A grande importância do "hash" universal traduz-se na palavra "qualquer" sublinhada no seguinte teorema.

Teorema 22 *Se H é universal então, para qualquer conjunto $N \subseteq U$ e para qualquer $x \in U$, o valor esperado, relativamente a uma escolha uniforme $h \in_U H$, do número de colisões entre x e*

[4] Relativamente à aleatorização do algoritmo determinada pela escolha dos pivots.

os outros elementos de N não excede n/a (número de elementos de N a dividir pelo tamanho da tabela de "hash").

Dem. Seja $y \in S$ com $x \neq y$. Definimos as seguintes variáveis aleatórias

$$\begin{cases} c_{xy} \text{ com valor 1 se } x \text{ e } y \text{ colidem e 0 caso contrário} \\ c_x \text{ número de colisões com } x\text{: número de elementos } y\text{, diferentes de } x\text{, tais que } h(y) = h(x). \end{cases}$$

Note-se que para o "hash" universal é

$$\begin{aligned} E(c_{xy}) &= 1 \times \text{prob}_{\,h \in_U H}[h(x) = h(y)] + 0 \times \text{prob}_{\,h \in_U H}[h(x) \neq h(y)] \\ &= \text{prob}_{\,h \in_U H}[h(x) = h(y)] \\ &\leq 1/a \end{aligned}$$

Temos $c_x = \sum_{y \in N, y \neq x} c_{xy}$. Usando em primeiro lugar a linearidade do valor esperado e depois a definição de "hash" universal vem

$$E(c_x) = \sum_{y \in N, y \neq x} E(c_{xy}) \leq n/a$$

$\square$

7.2.1 O método matricial de construção

O Teorema 22 traduz a vantagem do "hash" universal: tempo das operações básicas $O(1)$, independentemente dos dados. Mas será que o "hash" universal existe? Será que podemos construir eficientemente a correspondente família de funções de "hash"? Vamos ver que sim.

Escolha aleatória de uma matriz H

Por simplicidade, suponhamos que $a = 2^b$ (a dimensão da tabela de "hash" é uma potência de 2) e que $u = 2^d$ (a dimensão do universo é uma potência de 2). Vamos definir uma matriz H de dimensões $b \times d$; tipicamente H é larga e baixa, por exemplo, com 10 linhas e 1000 colunas. Cada elemento de H é, aleatoriamente e de forma uniforme, 0 ou 1.

Valor de $h(x)$

Seja $x \in U$; x tem d bits. O valor $h(x)$ é

$$h(x) = Hx$$

onde a multiplicação matricial é efectuada "módulo 2" (corpo $\{0, 1\}$).

Exemplo

Suponhamos que é $d = 5$, $b = 3$, $x = [0, 1, 0, 1, 1]^t$ e

$$h(x) = Hx = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Por exemplo, o segundo elemento de $h(x)$ é obtido a partir da segunda linha de H

$$(1 \times 0) + (0 \times 1) + (0 \times 0) + (1 \times 1) + (1 \times 1) = 0 + 0 + 0 + 1 + 1 = 0 \pmod 2$$

Note-se que uma maneira de ver este cálculo é o seguinte: somamos (módulo 2) alguns elementos da segunda linha de H. Quais? Os das linhas i para as quais $x_i = 1$. No exemplo acima, trata-se das linhas 2, 4 e 5 de x; isto é, temos a soma dos elementos de H de cor vermelha, $0 + 1 + 1 = 0 \pmod 2$.

Qual a probabilidade de ser $h(x) = h(y)$?

Consideremos agora dois elementos diferentes, x, $y \in U$, $x \neq y$. Como x e y são diferentes, têm que diferir pelo menos num bit, digamos no bit i; suponhamos então que é, por exemplo, $x_i = 0$ e $y_i = 1$. Vamos atribuir valores a todos os elementos de H, excepto aos da à coluna i. Notemos que, uma vez fixada essa escolha, $h(x)$ é fixo, uma vez que $x_i = 0$ (lado esquerdo da figura seguinte); por outro lado, cada uma das 2^b colunas i possíveis vai originar valores de $h(y)$ diferentes (no lado direito da figura está um desses valores). Para vermos isso, reparemos que, sendo $y_i = 1$, cada alteração do bit da linha j, $(1 \leq j \leq b)$ causa uma alteração do bit j de $h(y)$.

$$\overset{i}{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}} \times \overset{x}{\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}} = \overset{h(x)}{\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}} \qquad \overset{i}{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}} \times \overset{y}{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}} = \overset{h(y)}{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}$$

Concluimos que, sendo i um bit em que x e y diferem com $x_i = 0$ e $y_i = 1$, as 2^b colunas i possíveis

de H dão origem a 2^b valores de $h(y)$ distintos; assim[5], $h(x)$ e $h(y)$ são variáveis aleatórias independentes sendo $1/2^b = 1/a$ a probabilidade de ser $h(x) = h(y)$. Temos então o seguinte resultado

Teorema 23 *Seja $|A| = a = 2^b$ o tamanho da tabela de "hash" e $|U| = u = 2^d$ o tamanho do universo. Para $x \in U$ seja $h(x)$ definido por $h(x) = Hx$ onde H é uma matriz aleatória e uniforme de 0's e 1's com b linhas e d colunas, e o produto matricial é efectuado no corpo $\{0, 1\}$. Então, este conjunto de funções h é um conjunto universal de "hash".*

7.3 "Hash" perfeito

Com o "hash" universal temos uma garantia de que o *tempo médio* das operações básicas é $O(1)$. Mas o tempo no pior caso pode ser muito mau, nomeadamente $O(n)$. Vamos ver que para dicionários estáticos, utilizando o chamado "hash" perfeito, podemos construir eficientemente uma função de "hash" que garanta tempo $O(1)$, mesmo no pior caso.

Em princípio é desejável que o tamanho a da tabela de "hash" a utilizar não seja muito maior que o número n de elementos que estão lá armazenados, digamos que a deverá ser de ordem $O(n)$. Se permitirmos uma tabela de "hash" de tamanho n^2, a construção (através de um algoritmo aleatorizado) de uma tabela de "hash" perfeito é mais simples, como veremos em 7.3.1.

Se exigirmos uma tabela de "hash" com tamanho linear em n, tal construção também é possível, embora só tal só mais recentemente tenha sido conseguido, ver 7.3.2.

Comecemos por definir formalmente o "hash" perfeito.

Definição 12 *Uma função $h : U \to A$ diz-se uma função de "hash" perfeita para um conjunto dado $N \subseteq U$, se h restrita a N for injectiva (isto é, se não houver colisões entre os elementos de N).*

7.3.1 Construção com espaço $O(n^2)$

Se usarmos uma tabela de "hash" de tamanho n^2 é muito fácil construir uma função de "hash" perfeita usando um algoritmo aleatorizado de tempo polinomial. Com probabilidade pelo menos 1/2 esse algoritmo produz uma função de "hash" perfeita. Como sabemos do estudo dos algoritmos aleatorizados, ver página 65, pode conseguir-se, em tempo da mesma ordem de grandeza, uma probabilidade arbitrariamente próxima de 1.

[5] Relativamente à escolha aleatória dos elementos da coluna i de H.

Basicamente definimos uma função de "hash" aleatória (como no "hash" universal) e, como a tabela é grande, a probabilidade não existir qualquer colisão é grande.

Exercício 80 *São gerados k inteiros aleatórios, de forma unforme, entre 1 e n. Até que valor de k, a probabilidade de não existirem 2 inteiros gerados iguais é superior a 1/2?*

Exercício 81 *Numa sala estão 25 pessoas. Qual a probabilidade de pelo menos 2 dessas pessoas fazerem anos no mesmo dia? Calcule essa probabilidade de forma explícita (e não através de uma expressão). Use os seguintes axiomas: as pessoas nascem com igual probabilidade em todos os dias do ano; nenhum ano é bissexto.*

Teorema 24 *Seja $h(\cdot)$ uma função de "hash" definida através de uma matriz aleatória H correspondente a uma tabela de "hash" com n^2 elementos, construída como se descreveu em 7.2.1. A probabilidade de não haver qualquer colisão é pelo menos 1/2.*

Dem. Existem $\binom{n}{2}$ pares (x, y) de elementos do conjunto a representar. Por definição de "hash" universal, a probabilidade de colisão de um qualquer par específico é não superior a $1/a$. Portanto, a probabilidade p de existir pelo menos uma colisão satisfaz (note-se que $a = n^2$)

$$p \leq \binom{n}{2}/a = \frac{n(n-1)}{2n^2} < \frac{1}{2}$$

□

7.3.2 Construção com espaço $O(n)$

Durante bastante tempo duvidou-se que existisse um algoritmo eficiente de definir uma função de "hash" perfeita que mapeasse um conjunto dado de n elementos numa tabela de "hash" de $O(n)$ elementos. Após diversas tentativas, descobriu-se o seguinte método que é relativamente simples e elegante. Trata-se de um "hash" a 2 níveis. Note-se que o "hash" do primeiro nível não é (excepto se tivermos uma sorte excepcional) perfeito.

Construção do "hash" perfeito

1. Usando o "hash" universal define-se uma função h de "hash" (do primeiro nível) correspondente a uma tabela de "hash" de tamanho n, isto é, $a = n$.

2. Inserem-se todos os n elementos na tabela. Este "hash" não é perfeito, normalmente haverá diversas colisões (cadeias com mais que um elemento) bem como cadeias vazias.

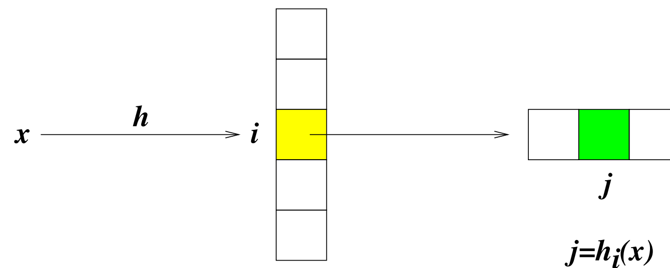
3. Num segundo nível de "hash", é construída para cada cadeia uma função de "hash" perfeito, segundo o algoritmo descrito em 7.3.1. Sejam h_1, h_2,..., h_n as respectivas funções de "hash" e A_1, A_2,..., A_n as respectivas tabelas. Note-se que a tabela A_i tem tamanho n_i^2 em que n_i é o número de elementos da cadeia i, isto é, elementos x com $h(x) = i$.

Antes de mostrar que este esquema aleatorizado produz com probabilidade pelo menos 1/2 um "hash" perfeito, vamos descrever o algoritmo de pesquisa.

Algoritmo de pesquisa

Dado x, retorna `V` of `F`.

1. Calcula-se $i = h(x)$
2. Calcula-se $j = h_i(x)$
3. Se $A_i[j]$ está ocupado com x, retorna-se `V`, senão (se estiver vazio ou contiver um valor diferente de x) retorna-se `F`



Observação. Vamos mostrar que este algoritmo é $O(1)$.

Observação. As cadeias – elementos com o mesmo valor $h(x)$ – têm muitas vezes 1 ou nenhum elemento; assim, é preferível com vista a poupar espaço e tempo, não construir tabelas de "hash" para esses casos, tratando-os de modo especial.

Correcção do algoritmo

Já verificamos que as funções de "hash" do segundo nível podem ser eficientemente construídas segundo o método de espaço quadrático, 7.3.1. Falta mostrar que o espaço gasto pelas tabelas A_i do segundo nível é $O(n)$.

Lema 2 *Se h é uma função uniformemente escolhida de um conjunto universal,*

$$\text{prob}\left\{\sum_{1\leq i\leq a} n_i^2 > 4n\right\} < \frac{1}{2}$$

Dem. Basta mostrar que $E(\sum_{1\leq i\leq a} n_i^2) < 2n$; na verdade seja a variável aleatória $Y = \sum_{1\leq i\leq a} n_i^2$. Pela desigualdade de Markov se $E(y) < 2n$, então prob $(Y > 4n) < 1/2$.

Se contarmos o número total de pares que colidem no "hash" do primeiro nível, incluindo as colisões dos elementos consigo próprios, vamos ter $\sum_{1\leq i\leq a} n_i^2$. Por exemplo, se a, b e c são os elementos de uma das cadeias, temos $3^2 = 9$ colisões que correspondem a todos os pares (x, y) com x e y pertencentes a $\{a, b, c\}$. Assim, e definindo-se c_{xy} como sendo 1 se x e y colidem e 0 caso contrário, temos que o valor médio da soma dos quadrados é

$$\begin{array}{rcll} E(\sum_i n_i^2) & = & \sum_x \sum_y E(c_{xy}) & \\ & = & n + \sum_x \sum_{y\neq x} E(c_{xy}) & \\ & \leq & n + n(n-1)/a & \text{porque } h \text{ é universal} \\ & \leq & n + n(n-1)/n & \text{porque } a = n \\ & < & 2n & \end{array}$$

□

7.4 Contar o número de elementos distintos

Consideremos o problema de determinar de forma eficiente quantos elementos distintos[6] existem numa longa sequência de, por exemplo, "strings".

Uma solução é usar um método de "hash", não inserindo um elemento quando ele já se encontra na tabela. No fim, conta-se quantos elementos existem na tabela.

Mas suponhamos agora que a sequência é realmente muito grande e que não dispomos de espaço suficiente para ter uma tabela de "hash". Por outro lado, aceitamos obter apenas um valor aproximado do número de elementos distintos. Por exemplo, podemos observar um "router" durante uma hora e pretendemos saber aproximadamente quantos IP's distintos existiram nas mensagens que foram encaminhadas pelo "router" nesse intervalo de tempo.

Uma solução é gerar uma função h de "hash" que produza um resultado uniforme em $[0, 1]$ (intervalo real) e determinar $\min_x h(x)$; se colocarmos de forma aleatória e uniforme p pontos $h(x_i)$ em $[0, 1]$, o valor médio de $\min_x h(x)$ é $1/(p + 1)$. Assim, calculamos esse mínimo e, a partir dele, o valor aproxmado de p. Obviamente, aqui não existe tabela de "hash", mas o método é inspirado nas funções de "hash".

Referências. Os leitores interessados na relação entre o hash universal, a criptografia e a "desa-leatorização" de algoritmos podem consultar por exemplo "Pairwise Independence and Derandomization" de Michael Luby e Avi Wigderson, acessível de

`http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/CSD-95-880.pdf`

[6] Se não houvesse a imposição de os elementos serem distintos, a solução era trivial...