Hashing na Solução de Problemas Atípicos

Lucila Maria de Souza Bento, Vinícius Gusmão Pereira de Sá
Departamento de Ciência da Computação – Instituto de Matemática – Universidade Federal do Rio de Janeiro (UFRJ)
21941-590 – Rio de Janeiro – RJ - Brasil
lucilabento@ppgi.ufrj.br, vigusmao@dcc.ufrj.br

Jayme Luiz Szwarcfiter
Instituto de Matemática – NCE e COPPE – Universidade Federal do Rio de Janeiro (UFRJ)
21941-916 – Rio de Janeiro – RJ - Brasil
Jayme@nce.ufrj.br

RESUMO

A organização de dados em tabelas de dispersão é uma técnica bastante difundida, onde os valores a serem persistidos estão relacionados a chaves de busca usadas para o cálculo dos endereços de seu armazenamento. Por proverem bom desempenho nas chamadas operações de dicionário (inserção, busca e remoção) sem requerer espaço exorbitante, as tabelas de dispersão são empregadas frequentemente na indexação de grandes volumes de dados e em aplicações típicas relacionadas à associação, busca e manipulação da informação. No entanto, há problemas que, mesmo possuindo natureza bem diversa, podem ser solucionados de maneira elegante com o emprego de tabelas de dispersão, resultando em sensível economia de tempo e espaço. Este artigo ilustra o poder dessa técnica, também chamada *hashing*, em situações que não lhe são comumente associadas.

PALAVARAS CHAVE. Tabelas de Dispersão, Complexidade, Algoritmos Eficientes

Área principal: OC – Otimização Combinatória

ABSTRACT

The organization of data into hash tables is a well-known technique in which the persistent values correspond to search keys upon which their storage addresses are calculated. Since they provide good performance for the so-called dictionary operations (insertion, lookup, deletion) without requiring exorbitant space, hash tables are very often employed in the indexation of large amounts of data and in typical applications related to search and handling of information. Nevertheless, there are problems of rather different nature that can be solved in elegant fashion by employing hash tables, yielding significant economy of time and space. This paper illustrates the power of such technique in situations it is not usually associated to.

KEYWORDS. Hash Tables, Complexity, Efficient Algorithms

Main area: OC – Combinatorial Optimization

1. Introdução

A técnica conhecida como *hashing* pode ser considerada um método de pesquisa e organização física de tabelas. Basicamente, utiliza-se uma função, chamada *função de dispersão* (ou *função hash*), para mapear identificadores (ou *chaves*) de um universo qualquer relativamente grande em números inteiros de um intervalo relativamente pequeno (Lewis, et al., 1988; Maurer, et al., 1975). Esses números indicarão as posições, em uma tabela, nas quais os dados (ou *valores*) correspondentes àquelas chaves deverão ser armazenados.[1] Tal tabela é chamada *tabela de dispersão* (ou *tabela hash*).

Idealmente, uma função de dispersão deveria mapear cada chave em uma posição distinta da tabela, o que é conseguido, em alguns casos, por meio de técnicas relacionadas ao chamado *hashing perfeito* (Dietzfelbinger, 2007). Na prática, nem sempre é possível definir tal função, de forma que chaves diferentes podem ser mapeadas em uma mesma posição, numa situação que é conhecida como *colisão*.

Quando ocorrem colisões, é necessário armazenar as chaves colididas em posições alternativas definidas por algum método de tratamento de colisões (Knuth, 1973). Dentre os mais utilizados, um dos mais simples é o método do *encadeamento exterior*, no qual as chaves são armazenadas em listas encadeadas que são acessadas de cada posição da tabela. É fácil ver que o número médio de elementos em cada lista encadeada é igual ao *fator de carga* da tabela, que, para uma tabela com m posições e n chaves armazenadas, é definido como $\alpha = n/m$ (Szwarcfiter, et al., 1994).

A complexidade das operações de inserção, busca e remoção em tabelas de dispersão depende da função *hash* utilizada e do fator de carga da tabela. Com o uso de uma função de espalhamento que disperse as chaves de forma aproximadamente uniforme pelas posições disponíveis, e com um fator de carga constante (o que é conseguido por um dimensionamento da tabela proporcional à quantidade de chaves a serem armazenadas), pode-se mostrar que tais operações são executadas em *tempo médio* $O(1)$ (Cormen, et al., 2001). Este desempenho esperado ótimo na recuperação da informação faz com que este método seja amplamente utilizado em tarefas de busca. Suas aplicações incluem a implementação de bancos de dados, tabelas de símbolos de compiladores e os mais variados tipos de dicionários.

Neste trabalho, ilustramos como a técnica do *hashing* pode ser utilizada na obtenção de soluções eficientes para problemas que intuitivamente não sugerem seu uso. Acreditamos que os exemplos apresentados – um problema aritmético, um algébrico, um de teoria de conjuntos e um de manipulação de arquivos – são suficientemente elucidativos, dispensando comentários gerais que os pretendessem envolver numa mesma categoria, ou que tentassem tipificar precisamente os casos em que tal técnica deva ser considerada.

2. Números *k*-complementares

Seja k um inteiro positivo, e seja A uma lista não-ordenada contendo $|A| = n$ números inteiros. O problema dos números k-complementares consiste em determinar todos os pares não-ordenados $\{x, y\}$ de elementos de A tais que a igualdade $x + y = k$ seja satisfeita. Seja, por exemplo, $A = [1, -4, 18, 11, 2, 9, -3, 5, 560, 10]$ e $k = 20$. A saída esperada compreende os pares $\{2, 18\}, \{9, 11\}$ e $\{10, 10\}$.

Em uma primeira solução ingênua, cada elemento da lista A é somado a cada um dos demais elementos, de forma que todos os pares de elementos de A sejam investigados, e localizados aqueles que somem k. Procedendo desta forma, o problema certamente será resolvido, mas, embora nenhum espaço extra seja requerido, a quantidade de adições que serão efetuadas será $O(n^2)$, resultando num algoritmo de tempo quadrático no tamanho da entrada.

[1] Em muitos casos, deseja-se armazenar tão somente as próprias chaves, sem que qualquer outro valor lhes esteja associado.

Queremos fazer melhor. Primeiramente, algo interessante a ser observado é que, se selecionamos um elemento x de A e calculamos seu k-complemento $y = k - x$, o próximo passo consistirá em simplesmente procurarmos y, ou seja, nesse instante o problema passaria a ser um problema de busca. Gostaríamos, nesse caso, de poder buscar um elemento rapidamente.

Uma possibilidade seria utilizarmos vetores (ou *array*s) booleanos para representarmos os elementos de A: cada elemento de A indicaria uma posição preenchida com um bit 1, sendo todas as demais posições do vetor preenchidas com bits 0. Na técnica conhecida como *endereçamento direto*, a indicação da posição de cada elemento é conseguida fazendo com que o *valor* do elemento constitua seu próprio *endereço*, isto é, o índice de sua posição no vetor que o armazenará. Dessa forma, o elemento 18 ficaria armazenado na 18ª posição do vetor; o elemento 560, na 560ª posição; e assim por diante.[2]

Há, no entanto, um problema grave nesta abordagem: o tamanho dos vetores deverá ser proporcional não ao tamanho de A, mas sim ao intervalo de valores possíveis que podem figurar em A, isto é, ao tamanho *do universo* de onde irão advir os valores da lista de entrada do problema. Se a lista contém, digamos, cem elementos positivos, o maior dos quais igual a um milhão, seria necessário um vetor com um milhão de posições para representar a lista! Mais formalmente, se w é um parâmetro da entrada do problema, indicando o tamanho do universo de valores que poderão constar na lista, então o espaço utilizado pelo algoritmo seria exponencial no tamanho ocupado por este parâmetro na especificação da instância de entrada (uma vez que w é representado por $\log w$ bits).

Neste ponto nos remetemos ao emprego de *hashing*. Utilizando uma tabela de dispersão para armazenar os n elementos de A, o espaço necessário será igual a $m = n/\alpha$. Para um fator de carga α constante, o tempo médio das operações de dicionário é também, como já mencionado, $O(1)$.

Já estamos diante de um algoritmo linear em espaço e em tempo médio para o problema dos números k-complementares. Percorre-se a lista A, armazenando seus elementos numa tabela de dispersão H inicialmente vazia. A seguir, percorre-se A uma segunda vez: para cada elemento x de A, busca-se em H o k-complemento y de x, imprimindo $\{x, y\}$ caso y esteja em H.

É claro que soluções utilizando *hashing* exigem uma elaboração um pouco maior na escrita do código do que soluções que utilizam apenas estruturas de dados mais simples como vetores e listas. Felizmente, a imensa maioria das linguagens de programação modernas de alto nível já oferecem – ou dispõem de bibliotecas que o fazem – implementações bastante eficientes de tabelas de dispersão, de forma que o programador não precisa necessariamente codificar as operações básicas para este tipo de estrutura, mas apenas utilizá-las, transparentemente, em seu proveito. De qualquer modo, para efeito de completude, apresentamos a seguir uma possível codificação do algoritmo proposto que dispensa qualquer suporte a *hashing* dado pela linguagem a ser utilizada.

Não é do escopo deste trabalho a concepção de funções *hash* apropriadas a cada caso, assunto este que requer ferramental estatístico e estudo específico (Knott, 1975). A partir daqui, suporemos a existência de uma função *hash* que mapeie valores x de entrada em números inteiros do intervalo $[0, m-1]$, onde m é o tamanho da tabela de dispersão utilizada.[3] Um exemplo possível seria a função muito simples $hash(x) = x \bmod m$ (Knuth, 1973).

A solução proposta está dividida em três partes: um procedimento para a criação e inicialização da estrutura básica, a tabela de dispersão; um procedimento para a inserção dos elementos de A na tabela; e, por fim, o algoritmo propriamente dito, que soluciona o problema dos números k-complementares.

---

[2] O fato de poder haver números negativos em A não chegaria a ser um problema, pois poderíamos usar dois vetores: um para os elementos não-negativos, outro para os elementos negativos de A, valendo-nos do módulo de cada número para a indicação de seu endereço no vetor correspondente.

[3] De fato, é possível encontrar boas funções de dispersão disponíveis nas modernas linguagens de programação.

O procedimento $inicializa\_hash()$ cria uma tabela de dispersão com encadeamento exterior, na forma de um vetor H com m posições. Cada posição i de H contêm um ponteiro para a lista encadeada L_i, na qual serão armazenadas as chaves mapeadas na posição i pela função *hash*. As atribuições realizadas nas linhas $3, 4$, 5 e 6 criam e inicializam as listas encadeadas associadas a cada posição de H. Cada uma destas atribuições é realizada em tempo $O(1)$, de forma que todo o procedimento roda em tempo $O(m)$.

procedimento: inicializa_hash(m)
entrada: inteiro m, indicando o tamanho desejado da tabela de dispersão
saída: tabela *hash* vazia
1. $H \coloneqq$ novo vetor de tamanho m
2. **para** $i \coloneqq 0$ **até** $m - 1$ **faça**
3. $L_i \coloneqq$ nova lista
4. L_i.primeiro $\coloneqq \emptyset$
5. L_i.ultimo $\coloneqq \emptyset$
6. $H[i] \coloneqq L_i$
7. **retorne** H

O procedimento *popula_hash*( ) transfere os elementos da lista de entrada para uma tabela de dispersão previamente inicializada. Note que cada iteração do laço do procedimento *popula_hash*( ) lê o próximo elemento da lista em tempo $O(1)$, calcula sua posição na tabela *hash* também em tempo $O(1)$ e, por fim, armazena o elemento no endereço-base correspondente, o que é feito alocando espaço e atualizando ponteiros, cada uma dessas operações também executada em tempo $O(1)$. Logo, como o laço é repetido para todo elemento de A, a complexidade do procedimento é $n.O(1) = O(n)$.

procedimento: popula_hash(A, H)
entrada: lista encadeada (ou vetor) A contendo n inteiros
tabela de dispersão H
saída: tabela H contendo os elementos de A
1. **para cada** elemento x de A **faça**
2. $h_x \coloneqq hash(x)$
3. $L_h \coloneqq H[h_x]$
4. $l \coloneqq$ novo item
5. l.chave $\coloneqq x$
6. l.proximo $\coloneqq \emptyset$
7. l.anterior $\coloneqq L_h$.ultimo
8. **se** L_h.primeiro = $\emptyset$ **então**
9. L_h.primeiro $\coloneqq l$
10. L_h.ultimo $\coloneqq l$
11. **retorne** T

Finalmente, o algoritmo $busca\_complementares()$ realiza a busca pelos pares $\{x, y\}$ que somam k. Neste algoritmo, o pior caso ocorre quando todos os elementos de A são mapeados para a mesma posição da tabela *hash*, de forma que uma busca nesta tabela corresponda a uma busca numa lista encadeada de tamanho n. A complexidade de pior caso do algoritmo é, portanto, $n \,.O(n) = O(n^2)$. No entanto, como se trata de uma solução com *hashing*, e como o espalhamento das chaves que é oferecida por boas funções *hash* se aproxima de uma distribuição aleatória e uniforme, o pior caso deve ser visto não como uma *entrada* que leva o algoritmo a executar o maior número possível de passos, mas como uma *função hash* que falha em distribuir os elementos de uma certa entrada de forma equilibrada. A probabilidade disto acontecer, no entanto, *para qualquer entrada fixa*, é em geral extremamente baixa. Em vista disto, a análise de

caso médio, nos algoritmos envolvendo *hashing*, fornece um indicador bastante fiel de seu desempenho prático, independentemente da natureza das instâncias de entrada que lhe venham a ser fornecidas.

algoritmo: busca_complementares(A, k)
entrada: lista encadeada A contendo n inteiros
inteiro k
saída: listagem de todo par $\{x, y\}$ tal que $x + y = k$
1. $m \coloneqq n/0.5$ // arbitrando um fator de carga de 0.5
2. $H \coloneqq inicializa\_hash(m)$
3. $H \coloneqq popula\_hash(A, H)$
4. **para cada** elemento *x* de A **faça**
5. $y \coloneqq k - x$
6. $h_y \coloneqq hash(y)$
7. $L_h \coloneqq H[h_y]$
8. $pt \coloneqq L_h$.primeiro
9. **enquanto** $pt \neq \emptyset$ **faça**
10. **se** pt.chave $= y$ **então**
11. imprima $\{x, y\}$
12. $pt \coloneqq \emptyset$
13. **senão**
14. $pt \coloneqq pt$.proximo;

Apresentamos, a seguir, um limite superior para o tempo médio de execução do algoritmo. Faremos isto considerando apenas entradas onde não há qualquer par de números k-complementares, de forma que cada uma das buscas pelo *k*-complemento *y* de um número *x* da lista de entrada tenha que exaurir toda a lista encadeada correspondente ao endereço-base de *y* na tabela de dispersão. Certamente, entradas que possuam pares de números k-complementares acarretarão certo número de buscas bem-sucedidas, buscas estas que serão concluídas possivelmente antes (mas nunca depois) que toda a lista correspondente tenha sido exaurida, de forma que o tempo gasto nestes casos será majorado pelo tempo gasto naqueles em que não há qualquer par de números *k*-complementares.

Se a entrada não possui números *k*-complementares, então, para cada elemento x_i da lista de entrada, o tempo T_i para se buscar sem sucesso seu k-complemento $y_i = k - x_i$ é uma variável aleatória cujo valor será igual ao tempo de acesso ao endereço-base de y_i, mais o tempo para se percorrer a lista encadeada apontada, na tabela de dispersão, por aquele endereço-base. Como, na média, o tamanho de uma lista é igual ao fator de carga α da tabela, temos $E[T_i] = O(1) + O(\alpha) = O(1)$, para α constante.

Para o tempo total T do algoritmo, escrevemos

$$T = \sum_{i=1}^{n} T_i \,,$$

e então, pela linearidade da esperança,

$$E[T] = E[\sum_{i=1}^{n} T_i] = \sum_{i=1}^{n} E[T_i] = n \,.\, O(1) = O(n).$$

Concluímos assim que, usando espaço linear no tamanho da entrada, o algoritmo proposto resolve o problema em tempo esperado linear, mesmo para entradas que não possuam qualquer par de números k-complementares. No algoritmo ingênuo analisado anteriormente, o

tempo de pior caso era quadrático, e este limite é justo, por exemplo, nos casos de entradas sem pares k-complementares.

3. Interseção de Conjuntos

O problema da interseção de conjuntos consiste exatamente no que seu nome indica: deseja-se encontrar elementos que pertençam simultaneamente a dois conjuntos A e B. Há diversas formas de resolvê-lo, sendo a mais simples provavelmente aquela em que, para cada um dos elementos x de um dos conjuntos, percorre-se possivelmente todo o outro conjunto à medida que seus elementos vão sendo comparados com x. Durante esse processo, imprimem-se os protagonistas de comparações bem-sucedidas. Sendo n_A e n_B os tamanhos de A e B, respectivamente, são então realizadas $n_A . n_B$ comparações no pior caso. Se fizermos $n = \max\{nA, nB\}$, podemos descrever a complexidade de tempo de tal algoritmo como sendo $On2$, que é o dobro do custo necessário para simplesmente percorrer todos os elementos de A e B uma única vez. Este fato nos leva a pensar em uma nova solução utilizando *hashing*.

A ideia é criar uma tabela de dispersão H e populá-la usando os elementos de um dos vetores de entrada (digamos o menor dos vetores, para economizarmos espaço, e chamemo-no de A, sem perda de generalidade). A seguir, os elementos de $A \cap B$ são obtidos percorrendo-se todas as posições do vetor B e comparando-se cada elemento x de B com os elementos y que estão armazenados em H no mesmo endereço-base que teria x, isto é, comparamos x e y quando $hash(x) = hash(y)$, onde $hash(\,)$ é a função utilizada para popular H (que ora omitimos pelos motivos já expostos, mas que pode ser tão simples quanto aquela apresentada na Seção 2). O algoritmo $interseção\_hashing(\,)$ é dado a seguir em pseudocódigo, e o tratamento de colisões é feito por encadeamento exterior.

algoritmo: interseção_hashing(A, B)
entrada: vetores A e B, com $|A| \leq |B|$
saída: $A \cap B$
1. $n_A := |A|$; $n_B := |B|$
2. $m := n_A$ / 0.5 // arbitrando um fator de carga de 0.5
3. $H := inicializa\_hash(m)$
4. $H := popula\_hash(A, H)$
5. **para** $i := 0$ até $n_B - 1$ **faça**
6. $x := B[i]$
7. $h_x := hash(x)$
8. $L_x := H[h_x]$
9. $pt := L_h$.primeiro
10. **enquanto** $pt \neq \emptyset$ **faça**
11. **se** pt.chave $= x$ **então**
12. imprime x
13. $pt := \emptyset$
14. **senão**
15. $pt := pt$.proximo;

Os procedimentos $inicializa\_hash(\,)$ e $popula\_hash(\,)$ correspondem aqui aos procedimentos homônimos da Seção 2.

Uma análise análoga à que fizemos na Seção 2 nos mostra que o número médio de operações efetuadas pelo algoritmo é igual à quantidade de elementos do maior conjunto vezes $(1 + \alpha)$, onde α é o fator de carga da tabela de dispersão. Como, ao utilizarmos espaço proporcional à quantidade de elementos a serem armazenados, estabelecemos α como uma constante, temos que a quantidade de comparações do algoritmo $interseção\_hashing(\,)$ é $O(n_B) . \alpha = O(n_B)$. Esta quantidade de comparações, cada uma das quais realizada em tempo

constante, somada às $O(n_A)$ operações para se popular a tabela com os elementos do menor conjunto, resulta numa complexidade total esperada de $O(n_A + n_B) = O(n)$ para o algoritmo. Trata-se, portanto, de um ganho considerável em relação aos tempos respectivos $O(n^2)$ e $O(n \log n)$ dos algoritmos anteriores.

4. Igualdade da Soma de Funções

O problema da igualdade da soma de funções consiste em encontrar as quádruplas ordenadas (x, y, z, w) formadas por elementos de certo conjunto A dado, satisfazendo $f(x) + f(y) = f(z) + f(w)$, para uma função real f qualquer avaliada em tempo constante. Este problema pode ser considerado um caso particular do problema da partição, que é NP-completo (Hayes, 2002), e no qual deseja-se saber se um conjunto de entrada S pode ser dividido em dois conjuntos S_1 e S_2 de mesma soma, não havendo, contudo qualquer restrição ao tamanho dos dois conjuntos.

A partir deste ponto, não nos preocuparemos mais com a inicialização das tabelas de dispersão, ou com as funções *hash*, ou os métodos de tratamento de colisão utilizados, que suporemos disponíveis e apropriados.

A abordagem trivial do problema exposto examinaria cada uma das quádruplas de elementos do conjunto A, para uma complexidade de tempo igual a $O(n^4)$, onde $n = |A|$. Como transformar este problema num problema de busca em que possamos nos valer da eficiência média das tabelas de dispersão?

Uma resposta adequada seria: armazenando os possíveis valores $d = f(x) + f(y)$, para todo $x, y \in A$, como chaves de uma tabela de dispersão H, cada uma das quais associadas a uma lista que conterá todos os pares ordenados (x, y) cujas imagens por f somem d. A seguir, para cada chave d armazenada em H, combinamos dois a dois os pares (x, y) pertencentes à lista associada a d, obtendo assim as quádruplas desejadas. O pseudocódigo do algoritmo $soma\_de\_funções\_hashing()$ é dado a seguir.

algoritmo: soma_de_funções_hashing(A, f)
entrada: vetor A
função f
saída: quádruplas ordenadas (x, y, z, w) satisfazendo $f(x) + f(y) = f(z) + f(w)$
1. $H :=$ tabela $hash$ inicialmente vazia
2. **para cada** $(x, y) \subseteq A$ **faça**
3. $d = f(x) + f(y)$
4. **se** d é chave de H **então**
5. $L_d :=$ lista encadeada associada a d, em H
6. **senão**
7. $L_d :=$ nova lista encadeada
8. insira em H a chave d e, associada a ela, a lista L_d
9. acrescente o par (x, y) à lista L_d
10. **para cada** d em H **faça**
11. $L_d :=$ lista encadeada associada a d, em H
12. **para cada** $(x, y) \in L_d$ **faça**
13. **para cada** $(z, w) \in L_d$ **faça**
14. imprima (x, y, z, w)

O espaço utilizado será aquele necessário para armazenar $O(n^2)$ pares numa tabela de dispersão, isto é, $O(n^2)$, para um fator de carga constante.

Quanto ao tempo, cada chave d pode ser localizada e/ou inserida em H em tempo médio constante, e cada um dos $O(n^2)$ pares de elementos (x, y) de A demanda tempo $O(1)$ para ser inserido no final da lista associada à chave $d = f(x) + f(y)$. Dessa forma, toda a tarefa de

se popular a tabela com cada chave d associada aos pares de elementos cujas imagens somem d pode ser executada em tempo médio $O(n^2)$.

Resta listar as quádruplas que satisfazem a igualdade desejada (linhas 10-14), o que é conseguido pelas diferentes combinações dos pares armazenados em uma mesma lista. Como é gasto tempo constante para cada par impresso, e sendo d a quantidade de tais pares, o tempo esperado de todo o algoritmo será $O(n^2 + d)$.

5. Problema do Mercado Financeiro

Certa empresa X atua no mercado financeiro negociando papéis com um grande número de parceiros. Diariamente, diversas negociações são feitas, muitas das quais de forma automática por *softwares* especializados. Ao longo de um dia típico, um parceiro pode realizar novas negociações com X ou cancelar as negociações realizadas até aquele instante.

Cada negociação feita por X recebe um código de portfólio, que é um número inteiro associado àquela operação. Note que operações com parceiros distintos podem receber o mesmo código, e operações com um mesmo parceiro podem receber códigos distintos.

As operações efetuadas ao longo de um dia são armazenadas, uma por linha, em um arquivo. Cada linha compreende um sinal de "+" ou de "−". Um sinal de "+" indica que uma nova negociação foi feita, e é sempre seguido do identificador de um parceiro comercial de X e do código de portfólio. Um sinal de "−", por outro lado, é seguido apenas do identificador de um parceiro comercial de X e indica o cancelamento de *todas* as negociações realizadas com aquele parceiro, desde o início do dia até o momento em que aquela entrada foi registrada no arquivo, não importando seus códigos de portfólio. Tanto os identificadores de parceiros quanto os códigos de portfólio são números inteiros advindos de um intervalo muito grande.

A Figura 1 mostra a estrutura básica desse arquivo de negociações.

Tipo	Parceiro	Portfólio
+	101	145113
+	2	26
+	101	26
+	3005	4550
−	101	
+	3005	145113
+	4	184
+	101	4550
−	2	

Figura 1 – arquivo de operações da empresa X

Diariamente, após o encerramento das negociações, a empresa X precisa detectar quais portfólios estão associados a uma ou mais negociações ativas, isto é, negociações que não foram canceladas. Qual a maneira mais eficiente de fazê-lo?

É fácil pensar em diversas formas de se processar o arquivo de operações para obter a informação desejada em tempo quadrático (ou pior) no tamanho do arquivo. Por exemplo, percorrendo-se o arquivo na ordem cronológica das entradas (isto é, de cima para baixo), podemos verificar, para cada linha indicando parceiro Y e portfólio P, se houve cancelamento das operações de Y em qualquer uma das linhas subsequentes, imprimindo P caso não tenha havido (e cuidando, evidentemente, para que o mesmo portfólio não seja impresso mais de uma vez). Examinaremos, no entanto, soluções mais eficientes utilizando *hashing*.

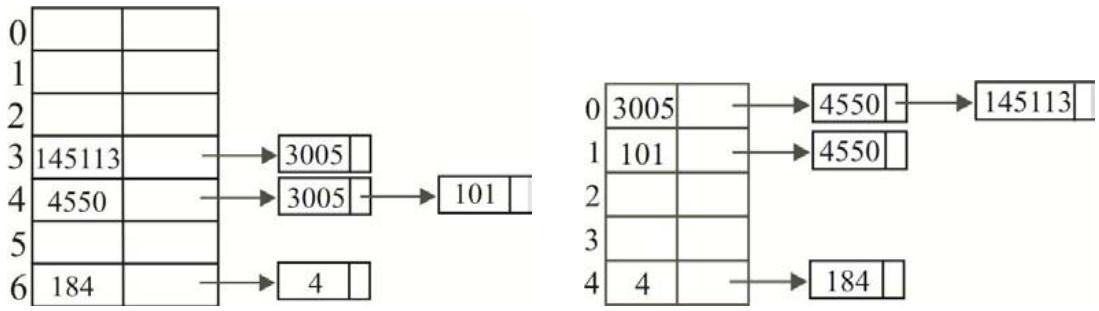
Considere a leitura do arquivo ainda sendo feita da primeira à última linha do arquivo. Utilizaremos duas tabelas de dispersão: uma, a de *parceiros por portfólio*, onde as chaves serão os códigos de portfólio, e os valores armazenados serão listas contendo os parceiros de X que tem negociações ativas sob aquele código de portfólio; e outra, a de *portfólios por parceiro*, onde as chaves serão os parceiros de X, e os valores associados a cada chave serão listas contendo os códigos de portfólio utilizados em negociações ativas com aquele parceiro.

Cada linha lida do arquivo com o sinal "+" acarretará duas alterações nas estruturas de dados utilizadas: a primeira é a inclusão de Y na lista de parceiros associados à chave P na tabela *hash* de parceiros por portfólio (a chave P será inserida pela primeira vez na tabela, caso lá ainda não se encontre); a segunda alteração é a inserção de P na lista de portfólios associados à chave Y na tabela de portfólios por parceiro (a chave Y será inserida pela primeira vez na tabela, caso lá ainda não se encontre).

Quando uma linha com o sinal "−" é lida para um parceiro Y, é preciso excluir as ocorrências de Y de ambas as tabelas. Na tabela de portfólios por parceiro, Y é chave e, pode, portanto, ser recuperada e removida em tempo médio $O(1)$. Na tabela de parceiros por portfólio, no entanto, Y pode pertencer a listas associadas a vários portfólios. Para não ser necessário percorrer todas as listas, usamos a informação da tabela de portfólios por parceiro (antes da exclusão da chave Y, evidentemente), para já saber, de antemão, quais são os portfólios em cujas listas se encontrará Y na tabela de parceiros por portfólio. Poderemos, assim, ir diretamente naquelas listas e remover Y, sem necessidade de percorrer toda a tabela.

Ao fim do processamento de todas as linhas do arquivo, basta retornar os portfólios que estiverem associados a listas não-vazias de parceiros, na tabela de parceiros por portfólio.

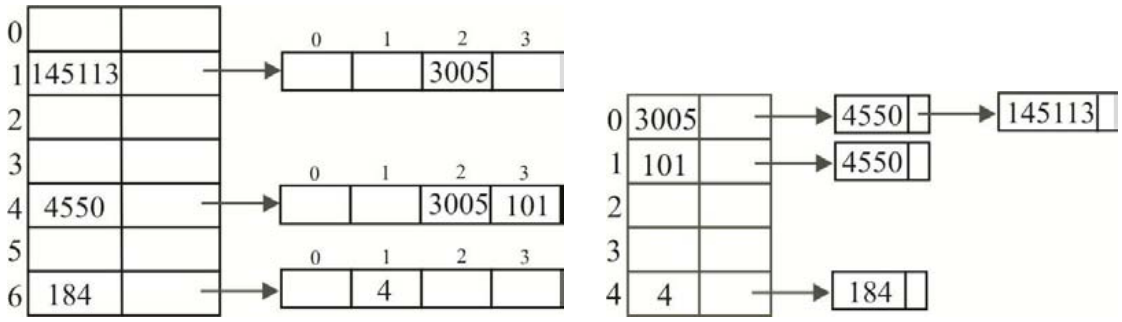
A Figura 2 ilustra as estruturas de dados utilizadas nesta solução proposta. Observe que, por clareza, não aparecem na figura chaves distintas com o mesmo endereço-base. Como mencionado anteriormente, deixamos o tratamento de colisões a cargo da implementação das tabelas *hash* utilizadas, liberando nosso foco para o que nos interessa, que é o emprego da técnica.



(a) **(b)**

Figura 2 – (a) parceiros por portfólio: tabela *hash* com listas
(b) portfólios por parceiro: tabela *hash* com listas

Como vimos, a localização de cada lista de parceiros da qual Y precisa ser removido pode ser feita em tempo médio $O(1)$. No entanto, para localizar Y dentro de cada uma das listas, precisaríamos percorrer toda a lista, o que certamente aumentaria o tempo computacional. Podemos, no entanto, substituir com vantagem tais listas por tabelas de dispersão! Em outras palavras, o valor associado a cada chave P na tabela *hash* original (parceiros por portfólio) será o ponteiro para uma tabela *hash* cujas chaves serão os parceiros que negociaram com a empresa X sob o código de portfólio P (veja a Figura 3). Sendo assim, diante de uma operação de cancelamento, a localização e a remoção de cada parceiro Y será feita também em tempo médio $O(1)$, e todo o algoritmo rodará então em tempo esperado constante no número de linhas do arquivo.



(a) **(b)**

Figura 3 – (a) parceiros por portfólio: tabelas *hash* aninhadas
(b) portfólios por parceiro: tabela *hash* com listas

A solução mais elegante para o problema do mercado financeiro, no entanto, consiste na leitura do arquivo de operações de trás para frente, isto é, da última para a primeira negociação realizada no dia, evitando, como mostraremos a seguir, o processamento de negociações que viriam a ser, de todo modo, posteriormente canceladas.

Note que, como o arquivo estará sendo lido de trás para frente, então a *primeira linha encontrada* com o sinal de "−", indicando o cancelamento das operações passadas com certo parceiro *Y*, é referente, na verdade, ao último cancelamento (cronologicamente) das operações com *Y* naquele dia. Dessa forma, essa linha particiona as demais linhas referentes a *Y* em dois grupos: as que ocorreram cronologicamente antes desse cancelamento (e que não foram ainda processadas, pois se encontram acima dela, no arquivo), e as que ocorreram cronologicamente após este cancelamento (e que, portanto, já foram processadas, uma vez que se encontram abaixo dela, no arquivo). A partir de então, as linhas do primeiro grupo poderão ser totalmente ignoradas pelo algoritmo, uma vez que as negociações correspondentes foram, de todo modo, canceladas. Por outro lado, as linhas referentes a *Y* que já foram processadas correspondem a negociações que não terão sido canceladas naquele dia, como é garantido pelo fato de aparecerem, no arquivo, após o último cancelamento envolvendo *Y*!

Assim, usaremos novamente duas tabelas de dispersão, ambas porém muito mais simples: uma para os portfólios ativos, que serão retornados no final; outra para indicar parceiros para os quais já foi processada uma linha indicando o último cancelamento de suas operações naquele dia. Com essa estrutura, cada linha com o sinal "−" verá o parceiro comercial em questão ser adicionado à tabela de parceiros cancelados (cujas operações serão ignoradas a partir de então); e cada linha com o sinal "+", *se* referente a um parceiro que ainda não foi cancelado, verá o portfólio em questão ser adicionado à tabela de portfólios ativos (de onde jamais sairá!); caso contrário, será simplesmente ignorada.

A Figura 4 mostra como as negociações da Figura 1 são armazenadas durante a execução do algoritmo proposto.

(a) **(b)**

Figura 4 – (a) portfólios ativos: tabela *hash* apenas com chaves
(b) parceiros cancelados: tabela *hash* apenas com chaves

Como tanto a busca de um parceiro na tabela de parceiros cancelados quanto a inserção de um portfólio na tabela de portfólios ativos podem ser realizadas em tempo médio $O(1)$, todo o processo demanda tempo esperado linear no tamanho do arquivo. Além disso, contando-se o número de parceiros cancelados e comparando-se esse número com o total de parceiros com os quais a empresa negociou ao longo do dia (informação esta que pode estar disponível), torna-se dispensável a leitura do restante do arquivo a partir do momento em que esses dois números se tornem iguais, uma vez que as demais linhas só irão conter negociações posteriormente canceladas. Dessa forma, o algoritmo pode parar antes mesmo que todas as linhas do arquivo tenham sido processadas.

5. Considerações Finais

O emprego de estruturas de dados apropriadas é um dos pontos principais a se considerar na elaboração de algoritmos eficientes. Ocorre que determinadas estruturas, e não é o caso apenas das tabelas de dispersão, ficam de certa forma estigmatizadas, restritas a determinadas aplicações clássicas e suas variantes. É importante, contudo, saber aliar as ferramentas teóricas à criatividade ao considerar o uso de estruturas eficientes, mesmo em aplicações que, à primeira vista, não pareceriam sugeri-las.

Neste trabalho, ilustramos o uso de técnicas envolvendo *hashing* na concepção de soluções simples e eficientes para problemas pouco ortodoxos, ou em soluções pouco ortodoxas, mas eficientes, para problemas simples. Acreditamos serem os exemplos apresentados, embora pouco numerosos, suficientemente elucidativos.

Referências

Cormen, T. H., et al. 2001. *Introduction to Algorithms.* s.l. : The MIT Press, 2001.
Dietzfelbinger, M. 2007. Design strategies for minimal perfect hash functions. *Lecture Notes in Computer Science*. 4665, 2007, 2-17.
Hayes, B. 2002. The Easiest Hard Problem. *American Scientist.* May-June de 2002, p. 113.
Knott, G. D. 1975. Hashing functions. *The Computer Journal*. 18, 1975, 265-278.
Knuth, D. E. 1973. *The Art of Computer Programming 3: Sorting and Searching.* Reading, Ma. : Addison-Westley, 1973.
Lewis, T. G. e Cook, C. R. 1988. Hashing for dynamic and static internal tables. *IEEE Computer.* 21, 1988, 45-56.
Maurer, W. D. e Lewis, T. G. 1975. Hash table methods. *ACM Computer Surveys.* 7, 1975, 5-19.
Szwarcfiter, J. L. e Markenson, L. 1994. *Estruturas de Dados e Seus Algoritmos.* s.l. : LTC, 1994.