

Continuação do Programa 5.36

```

TipoTodosPesos = array [Tipor] of Tipopesos;
Tipog           = array [0..MAXNUMVERTICES] of integer;
TipoChave       = packed array [1..MAXTAMCHAVE] of char;
TipoConjChaves = array [0..MAXNUMCHAVES] of TipoChave;
TipoIndice      = 0..MAXNUMARESTAS;

var
  M      : TipoValorVertice;
  N      : TipoValorAresta;
  r      : Tipor;
  g      : Tipog;
  Pesos  : TipoTodosPesos;
  i, j   : integer;
  ConjChaves: TipoConjChaves;
  NomeArq : string [100];
  Chave   : TipoChave;
  ArqEntrada: text;
{ Entra aqui a funcao hash universal do Programa 5.23 }
{ Entra aqui a funcao hash perfeita do Programa 5.35 }
begin
  write ( 'Nome do arquivo com chaves a serem lidas: ' );
  readln ( NomeArq );
  assign ( ArqEntrada, NomeArq );
  reset ( ArqEntrada );
  readln ( ArqEntrada, N );
  readln ( ArqEntrada, M );
  readln ( ArqEntrada, r );
  for j := 0 to r - 1 do
    begin
      for i := 1 to MAXTAMCHAVE do read ( ArqEntrada, Pesos [j][i] );
      readln ( ArqEntrada );
    end;
  for i := 0 to M-1 do read ( ArqEntrada, g[i] ); readln ( ArqEntrada );
  readln ( Chave );
  while Chave <> 'aaaaaa' do
    begin
      writeln (hp(Chave, r, Pesos, g));
      readln (Chave);
    end;
  close (ArqEntrada);
end. { hashingperfeito }

```

Análise A questão crucial é: quantas interações são necessárias para se obter um hipergrafo $G_r = (V, A)$ que seja acíclico? A resposta a esta questão depende dos valores de r e M escolhidos no primeiro passo do algoritmo. Obviamente, quanto maior o valor de M , mais esparsa é o grafo e, consequentemente, mais provável que ele seja acíclico. A influência do valor de r é discutida a seguir.

Segundo Czech, Havas e Majewski (1992, 1997), quando $M = cN$, $c > 2$ e $r = 2$, a probabilidade P_{ra} de gerar aleatoriamente um 2-grafo acíclico $G_2 = (V, A)$, para $N \rightarrow \infty$, é:

$$P_{ra} = e^{\frac{1}{c}} \sqrt{\frac{c-2}{c}}.$$

Por exemplo, quando $c = 2,09$ temos que $P_{ra} = 0,33$. Logo, o número esperado de iterações para gerar um 2-grafo acíclico é $1/P_{ra} = 1/0,33 \approx 3$. Isso significa que, em média, aproximadamente 3 grafos serão testados antes que apareça um 2-grafo acíclico para ser usado na geração da função de transformação. O custo para gerar cada grafo é linear no número de arestas do grafo. O procedimento GrafoAcíclico para verificar se um hipergrafo é acíclico do Programa 7.10 tem complexidade $O(|V| + |A|)$. Logo, a complexidade de tempo para gerar a função de transformação é proporcional ao número de chaves a serem inseridas na tabela hash, desde que $M > 2N$.

O grande inconveniente de usar $M = 2,09N$ é o espaço necessário para armazenar o arranjo g . Entretanto, existe outra maneira de aproximar o valor de M em direção ao valor de N . A alternativa é utilizar valores maiores de r . Majewski, Wormald, Havas e Czech (1996) mostraram, analítica e experimentalmente, que para 3-grafos o valor de M pode ser tão baixo quanto $1,23N$. Logo, o uso de 3-grafos reduz o custo de espaço da função de transformação perfeita, mas aumenta o tempo de acesso ao dicionário, pois requer o cômputo de mais uma função de transformação auxiliar h_2 e mais um acesso ao arranjo g .

Majewski, Wormald, Havas e Czech (1996) mostraram que o problema de gerar hipergrafos acíclicos randômicos para $r = 2$ e $r > 2$ têm naturezas diferentes. Para $r = 2$, a probabilidade P_{ra} varia continuamente com a constante c . Para $r > 2$, existe uma fase de transição: existe um valor $c(r)$ tal que se $c \leq c(r)$, então P_{ra} tende para 0 quando N tende para ∞ ; se $c > c(r)$, então P_{ra} tende para 1. Isso significa que, em média, um 3-grafo é obtido na primeira tentativa quando $c \geq 1,23$.

Uma vez obtido o hipergrafo, o procedimento Atribuir apresentado no Programa 5.30 é determinístico e requer um número linear de passos para rotular o hipergrafo e obter o arranjo g .

O número de bits por chave para descrever a função é uma medida de complexidade de espaço importante. Como cada entrada do arranjo g usa $\log N$ bits, a complexidade de espaço do algoritmo é $O(\log N)$ bits por chave, que é o espaço para descrever a função. De acordo com Majewski, Wormald, Havas e Czech (1996), o limite inferior para descrever uma função perfeita com ordem preservada é $\Omega(\log N)$ bits por chave, o que significa que o algoritmo que acabamos de ver é ótimo para essa classe de problemas. Na próxima seção vamos apresentar um algoritmo de hashing perfeito sem ordem preservada que reduz o espaço ocupado pela função de transformação de $O(\log N)$ para $O(1)$.