CAPÍTULO

14

ÁRVORES DE HUFFMAN

Mostraremos uma aplicação prática de árvores para compressão de arquivos, implementando operações para compressão e descompressão de cadeias armazenadas em vetores.

14.1 Fundamentos

Uma *árvore de Huffman* é uma árvore estritamente binária usada para *compressão* de arquivos, visando reduzir o espaço necessário para armazená-los em disco ou o tempo necessário para transmiti-los por um canal de comunicação.

Seja σ uma cadeia de n caracteres representando um *arquivo*. O *alfabeto* de σ, denotado por Σ, é o conjunto de caracteres que compõem σ. Por exemplo, para σ = "*Abracadabra!*", temos $n = 12$ e Σ = {*A*, *b*, *r*, *a*, *c*, *d*, !}. Então, supondo que os caracteres de Σ sejam codificados em ASCII, o *tamanho* de σ é $n \times 8 = 96$ bits.

Um modo de comprimir o arquivo σ é usar um *código de tamanho fixo* (CTF), em que cada um dos m caracteres do alfabeto Σ é codificado com $\lceil \lg m \rceil$ bits. Por exemplo, para σ = "*Abracadabra!*", cada caractere de Σ pode ser codificado com $\lceil \lg 7 \rceil = 3$ bits, como mostra a Figura 14.1: ! ↦ 000, *A* ↦ 001, *c* ↦ 010, d ↦ 011, *b* ↦ 100, *r* ↦ 101 e *a* ↦ 110. Com esse código, o arquivo σ (comprimido) passa a ter apenas $12 \times 3 = 36$ bits, ou seja, 37,5% do tamanho que ele tinha antes.

Outro modo de comprimir o arquivo σ é usar um *código de tamanho variável* (CTV), em que os caracteres menos frequentes em σ são codifi-

cados com mais bits e os mais frequentes são codificados com menos bits, como mostra a Figura 14.1: $! \mapsto 0000, A \mapsto 0001, c \mapsto 0010, d \mapsto 0011, b \mapsto 010, r \mapsto 011$ e $a \mapsto 1$. Assim, o número médio de bits por caractere pode ser reduzido. Com esse novo código, o arquivo σ (comprimido) passa a ter apenas $1 \times 4 + 1 \times 4 + 1 \times 4 + 1 \times 4 + 2 \times 3 + 2 \times 3 + 4 \times 1 = 32$ bits, ou seja, 33,3% do tamanho que ele tinha antes.

Caractere	Frequência	ASCII	CTF	CTV
!	1	00100001	000	0000
A	1	01000001	001	0001
c	1	01100011	010	0010
d	1	01100100	011	0011
b	2	01100010	100	010
r	2	01110010	101	011
a	4	01100001	110	1

Figura 14.1 | Diferentes códigos para representar os caracteres de σ = *"Abracadabra!"*.

De fato, como veremos a seguir, uma árvore de Huffman é uma estrutura que representa um código de tamanho variável, que minimiza o número de bits necessários para representar um arquivo. Diferentemente do código ASCII, que é padrão, o *código de Huffman* varia em função do arquivo que é comprimido (e, portanto, deve ser criado especificamente para cada arquivo). Ademais, o código de Huffman é *livre de prefixo*, isto é, para quaisquer dois caracteres α e β no alfabeto Σ, o código de α não é um prefixo do código de β. Essa propriedade garante que a descompressão de um arquivo σ pode ser feita sem ambiguidade.

14.1.1 O algoritmo de Huffman

Dada uma cadeia de caracteres σ, representando um arquivo a ser comprimido, uma árvore de Huffman correspondente pode ser construída do seguinte modo:

- Para cada caractere ASCII c, obtenha a frequência $f(c)$ de c em σ.
- Seja Σ o conjunto de caracteres ASCII c, tais que $f(c) > 0$.
- Para cada caractere c em Σ, crie uma árvore binária correspondente com uma folha contendo $f(c)$. O conjunto de árvores criadas nesse passo é uma *floresta*. O valor na raiz de cada árvore da floresta é o *peso* da árvore.
- Enquanto houver mais que uma árvore na floresta: remova duas árvores de pesos mínimos, digamos A_e e A_d; em seguida, crie uma árvore binária A cuja raiz guarde a soma dos pesos de A_e e A_d e cujos filhos sejam A_e e A_d; finalmente, insira a árvore A na floresta.

- No fim da repetição, a árvore que sobra é uma árvore de Huffman para o arquivo representado por σ. Para obter os códigos comprimidos, rotule as ligações à esquerda com 0 e as ligações à direita com 1. Depois disso, o código de Huffman para cada caractere *c* é a sequência de bits que rotula o caminho que vai da raiz da árvore até a folha associada ao caractere *c*.

  A Figura 14.2 ilustra os passos desse algoritmo para σ = "*Abracadabra*!".

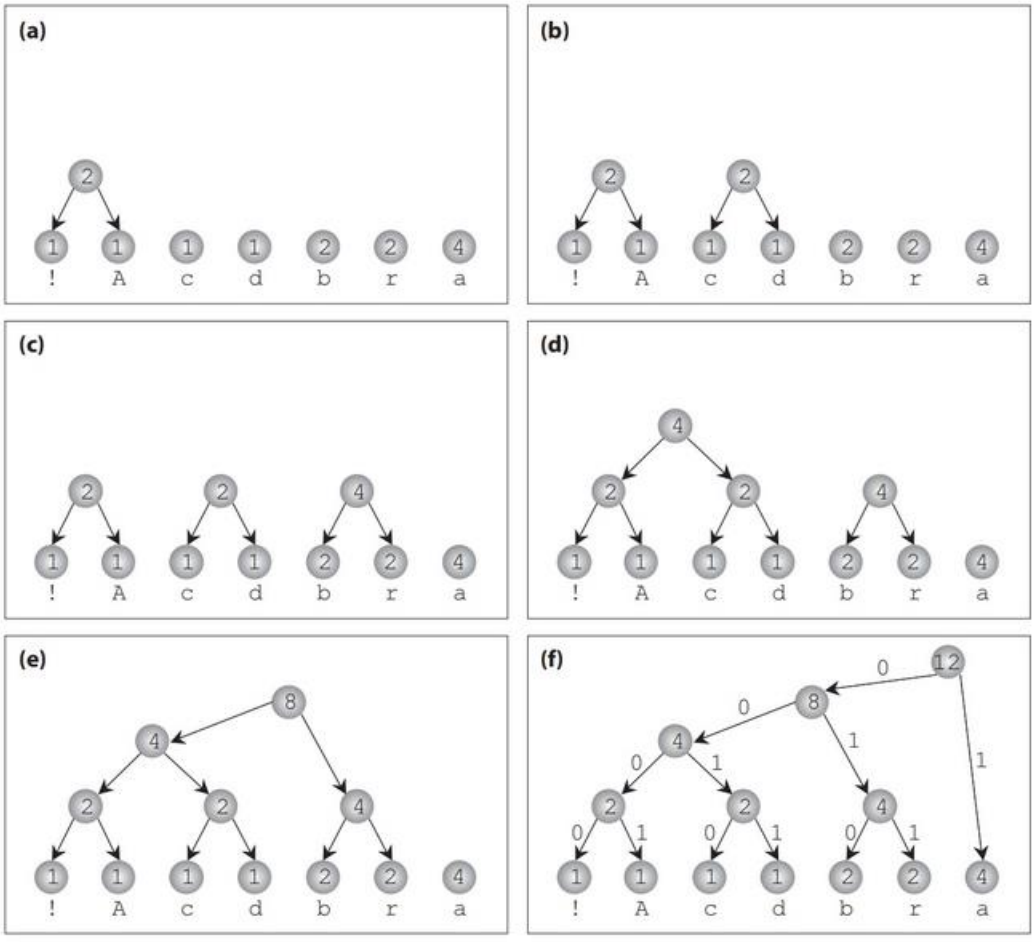


Figura 14.2 | Criação de uma árvore de Huffman para σ = "*Abracadabra*!".

14.2 Código de Huffman

Nessa seção, uma implementação do algoritmo de Huffman é apresentada.

14.2.1 Frequências dos caracteres

Dada uma cadeia de caracteres `s`, representando o conteúdo de um arquivo a ser comprimido, a função na Figura 14.3 devolve um ponteiro para um vetor de inteiros `f` tal que, para cada caractere ASCII `c`, `f[c]` é a frequência de `c` em `s`.

```
int *freq(char *s) {
   static int f[256];
   for(int i=0; i<256; i++) f[i] = 0;
   for(int i=0; s[i]; i++) f[s[i]]++;
   return f;
}
```

Figura 14.3 | Função para obtenção das frequências dos caracteres ASCII numa cadeia.

A função `freq()` usa um vetor estático `f` (isto é, que permanece na memória quando a execução da função termina), cujas posições são todas iniciadas com `0`. Conforme a cadeia `s` é percorrida, para cada caractere `s[i]`, o contador correspondente `f[s[i]]` é incrementado. Assim, no fim do percurso, o vetor `f` indica a frequência de cada caractere ASCII em `s`. Por exemplo, para `s = "A"` e `i = 0`, a instrução `f[s[i]]++` equivale a `f['A']++` (isto é, conta uma ocorrência de `'A'`).

14.2.2 Criação de árvore de Huffman

Para criar uma árvore de Huffman, vamos usar o tipo `Arvh`, definido na Figura 14.4, e a função `arvh()`, definida na Figura 14.5. Essa função recebe como entrada um *caractere*, sua *frequência* e duas *árvores* de Huffman e, como saída, ela devolve o endereço de um nó criado e preenchido com esses valores. Assim, cada folha numa árvore de Huffman guardará um caractere e sua frequência.

```
typedef struct arvh {
   struct arvh *esq;
   char         chr;
   int          frq;
   struct arvh *dir;
} *Arvh;
```

Figura 14.4 | Definições para criação de árvore de Huffman.

```
Arvh arvh(Arvh e, char c, int f, Arvh d) {
   Arvh n = malloc(sizeof(struct arvh));
   n->esq = e;
   n->chr = c;
   n->frq = f;
   n->dir = d;
   return n;
}
```

Figura 14.5 | Função para criação de um nó de árvore de Huffman.

Uma floresta será representada por um vetor `F` com `m` árvores de Huffman, em ordem *decrescente* de peso. Para garantir a ordenação da floresta, usaremos a função na Figura 14.6. Dadas uma árvore `A` e uma floresta `F` com `m` árvores, essa função insere a árvore `A` em `F`, em ordem decrescente de peso, e incrementa `m`. O parâmetro `m`, que indica o tamanho da floresta, é passado por referência.

```
void insf(Arvh A, Arvh F[], int *m) {
   int i = *m;
   while( i>0 && F[i-1]->frq < A->frq ) {
      F[i] = F[i-1];
      i--;
   }
   F[i] = A;
   (*m)++;
}
```

Figura 14.6 | Função para inserção ordenada de uma árvore na floresta.

Como uma floresta é um vetor de árvores em ordem decrescente de peso, para remover uma árvore de peso mínimo da floresta, basta remover a última árvore do vetor. Dada uma floresta `F` com `m` árvores, a função definida na Figura 14.7 decrementa `m` e devolve a última árvore de `F` como resposta.

```
Arvh remf(Arvh F[], int *m) {
   if( *m == 0 ) abort();
   return F[--(*m)];
}
```

Figura 14.7 | Função para remoção de uma árvore de peso mínimo da floresta.

A função `huffman()`, definida na Figura 14.8, cria uma árvore de Huffman correspondente a uma cadeia de caracteres `s`, que é dada como parâmetro.

```
Arvh huffman(char *s) {
   Arvh F[256];
   int  m = 0;
   int *f = freq(s);
   for(int c=0; c<256; c++)
      if( f[c]>0 )
         insf(arvh(NULL,c,f[c],NULL),F,&m);
   while( m>1 ) {
      Arvh d = remf(F,&m);
      Arvh e = remf(F,&m);
      insf(arvh(e,'-',e->frq+d->frq,d),F,&m);
   }
   return F[0];
}
```

Figura 14.8 | Função para criação de uma árvore de Huffman para uma cadeia `s`.

A função `huffman()` chama a função `freq()` para obter o vetor `f`, com as frequências dos caracteres ASCII em `s`. Em seguida, para cada caractere `c` que ocorre em `s` (isto é, tal que `f[c]>0`), ela cria uma folha contendo `c` e `f[c]` e insere essa folha na floresta `F`. A partir daí, enquanto houver mais que uma árvore em `F`, ela remove de `F` duas

árvores de pesos mínimos, cria um nó pai para elas, contendo a soma de seus pesos, e insere esse nó na floresta (note que, para nós que *não* são folhas, o valor do campo `chr` é irrelevante). No final, a árvore que sobra na floresta é a árvore de Huffman devolvida como resposta.

14.2.3 Exibição de árvore de Huffman

A função para exibição de uma árvore de Huffman, definida na Figura 14.9, usa um percurso *em-ordem* para visitar os nós da árvore e uma variável estática `n` para indicar o nível de cada nó na árvore. No início de cada chamada, o valor de `n` é incrementado e, no final, ele é decrementado. Então, para cada nó visitado, o valor `n` pode ser usado para endentar a exibição do nó (isto é, para produzir um espaçamento de `5` colunas por nível, antes de exibir os dados guardados no nó).

```
void exibeh(Arvh A){
   static int n = -1;
   if( A == NULL ) return;
   n++;
   exibeh(A->dir);
   for(int i=0; i<5*n; i++) printf(" ");
   printf("(%c,%d)\n",A->chr,A->frq);
   exibeh(A->esq);
   n--;
}
```

Figura 14.9 | Função para exibição de uma árvore de Huffman.

Por exemplo, a chamada `exibeh(huffman("Abracadabra!"))` produz a saída:

```
     (a,4)
(-,12)
               (r,2)
          (-,4)
               (b,2)
     (-,8)
                    (A,1)
               (-,2)
                    (!,1)
          (-,4)
                    (d,1)
               (-,2)
                    (c,1)
```

Essa saída corresponde à árvore de Huffman na Figura 14.10. Embora essa árvore seja distinta daquela na Figura 14.2f, o código representado por ela também minimiza o total de bits na cadeia compactada. De fato, uma cadeia pode ter várias árvores de Huffman (devido à ordem em que árvores de mesmo peso são escolhidas e à ordem em que elas atribuídas como filhas de um nó), mas todas elas geram uma cadeia comprimida correspondente de tamanho mínimo.

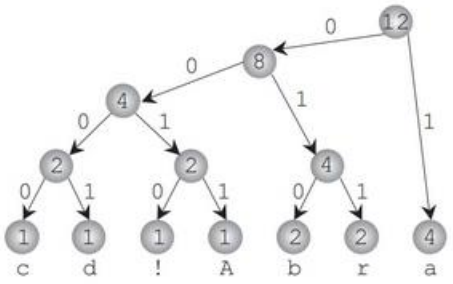


Figura 14.10 | Uma árvore de Huffman para s = "*Abracadabra!*".

14.2.4 Exibição do código de Huffman

A função que exibe o código representado por uma árvore de Huffman, definida na Figura 14.11, percorre a árvore *em-ordem* e processa apenas suas folhas.

```
void codigo(Arvh A) {
   static char d[256], t = -1;
   if( A == NULL ) return;
   if( A->esq == NULL && A->dir == NULL )
      printf("%c: %.*s\n",A->chr,t+1,d);
   else {
      t++;
      d[t] = '0'; codigo(A->esq);
      d[t] = '1'; codigo(A->dir);
      t--;
   }
}
```

Figura 14.11 | Função para exibição do código de Huffman.

Na função `codigo()`, o vetor estático `d` guarda uma cadeia de dígitos, criada conforme as chamadas recursivas são feitas: antes de uma chamada à esquerda, `d` é estendido com o dígito `'0'` e, antes de uma chamada à direita, `d` é estendido com o dígito `'1'`. Note que o vetor `d` funciona como uma pilha com topo `t` (isto é, que indica a posição do último dígito inserido em `d`). Então, quando uma folha é alcançada durante o percurso da árvore, o valor de `t` é usado em `printf()` para truncar a exibição da cadeia `d` (pois `d` não tem `'\0'`). O truncamento de cadeia é feito com o formato `"%.*s"`, que tem como parâmetros o tamanho da cadeia a ser exibida (isto é, `t+1`) e um ponteiro para cadeia (isto é, `d`).

Por exemplo, a chamada `codigo(huffman("Abracadabra!"))` produz a saída:

```
c: 0000
d: 0001
!: 0010
A: 0011
b: 010
r: 011
a: 1
```

14.3 Uso do código de Huffman

O código representado por uma árvore de Huffman serve para *comprimir* uma cadeia de caracteres `s` ou, então, para *descomprimir* uma cadeia de dígitos `d`.

14.3.1 Compressão de uma cadeia de caracteres

Para comprimirr uma *cadeia de caracteres* `s`, é preciso ter uma *tabela* de códigos, extraídos da árvore de Huffman `A`, criada a partir de `s`. Essa tabela é um vetor de `256` ponteiros para *cadeias de dígitos*, representando os códigos de Huffman.

A função que cria essa tabela, na Figura 14.12, é similar àquela na Figura 14.11; porém, em vez de exibir os códigos extraídos da árvore `A` no vídeo, ela os armazena no vetor `T`, recebido como parâmetro. A função `_strndup()`, declarada em `string.h`, é usada para fazer uma cópia da cadeia `c`, antes de guardá-la em `T`.

```
void tabela(Arvh A, char *H[]) {
   static char c[256], t = -1;
   if( A == NULL ) return;
   if( A->esq == NULL && A->dir == NULL )
      H[A->chr] = _strndup(c,t+1);
   else {
      t++;
      c[t] = '0'; tabela(A->esq,H);
      c[t] = '1'; tabela(A->dir,H);
      t--;
   }
}
```

Figura 14.12 | Função para criação de uma tabela de códigos de Huffman.

A compressão de uma cadeia é feita com a função definida na Figura 14.13. Essa função cria uma tabela `T`, com os códigos extraídos da árvore `A`, e para cada caractere `s[i]`, ela exibe o código binário correspondente, dado por `T[s[i]]`.

```
void comprimir(char *s, Arvh A) {
   char *T[256];
   for(int c=0; c<256; c++) T[c] = NULL;
   tabela(A,T);
   for(int i=0; s[i]; i++) printf("%s",T[s[i]]);
   for(int c=0; c<256; c++) free(T[c]);
}
```

Figura 14.13 | Função para comprimir uma cadeia de caracteres.

Por exemplo, a chamada `comprimir("Abracadabra!",A)` produz a saída:

```
00110100111000010001101001110010
```

14.3.2 Descompressão de uma cadeia de dígitos

Para descomprimir uma *cadeia de dígitos* `d`, basta ter a árvore de Huffman `A`, que foi usada para comprimi-la. A função na Figura 14.14 faz o ponteiro `n` apontar a raiz da árvore `A` e inicia `i` em `0`. A partir daí, de acordo com o valor de `b[i]`, ela faz `n` apontar a raiz da subárvore esquerda ou direita; se uma folha é alcançada, ela exibe o caractere guardado nessa folha e faz o ponteiro `n` voltar a apontar a raiz da árvore; em seguida, o valor de `i` é incrementado e o processo é repetido.

```
void descomprimir(char *d, Arvh A) {
   if( A == NULL ) return;
   Arvh n = A;
   for(int i=0; d[i]; i++) {
      n = (d[i]=='0') ? n->esq : n->dir;
      if( n->esq == NULL && n->dir == NULL ) {
         printf("%c",n->chr);
         n = A;
      }
   }
}
```

Figura 14.14 | Função para descomprimir uma cadeia de dígitos.

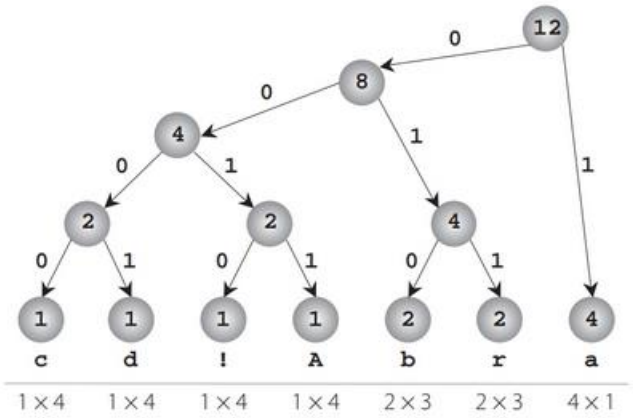
Por exemplo, `descomprimir("00110100111000010001101001110010",A)` gera a saída: `Abracadabra!`

Exercícios

14.1 Crie duas árvores de Huffman para a cadeia `"marmelada"`, como na Figura 14.2, e mostre que elas geram cadeias comprimidas de mesmo tamanho.

14.2 A *taxa de compressão* para uma árvore de Huffman `A`, criada a partir de uma cadeia `s`, é $\gamma = (1 - T_H/T_A) \times 100\%$, sendo T_H o total de bits na cadeia comprimida correspondente a `s` e T_A o total de bits na cadeia `s`, codificada em ASCII. Calcule a taxa de compressão para a cadeia `"marmelada"`, obtida com as árvores de Huffman criadas no exercício anterior.

14.3 A taxa de compressão para uma árvore de Huffman `A` pode ser calculada a partir dos dados representados na própria árvore `A`. O valor T_A é a frequência armazenada na raiz de `A`, multiplicada por `8` (isto é, `8*A->frq`). Para calcular T_H, multiplique a frequência armazenada em cada folha de `A` pelo comprimento `c`, do caminho que vai da raiz de `A` até ela, e some os resultados dos produtos, como ilustrado na figura a seguir.



(a) Crie a função `Th(A,c)`, que calcula T_H para a árvore de Huffman `A`. Essa função deve ser chamada com `c` igual a `0` e, a cada chamada recursiva, ele deve ser atualizado para `c+1`. Assim, quando uma folha de `A` for alcançada, `c` indicará o comprimento do caminho percorrido até ela. Use a seguinte estratégia recursiva: se A é uma folha, devolva o valor `c*A->frq`; senão, devolva a soma de `Th(A->esq,c+1)` e `Th(A->esq,c+1)`.

(b) Usando a função `Th()`, crie a função `taxa(A)`, que devolve a taxa de compressão obtida com a árvore de Huffman `A`.

(c) Crie um programa para testar a função `taxa()`. Por exemplo, a chamada `taxa(huffman("marmelada"))` deve devolver o valor `69.4`.

14.4 Usando as funções apresentadas nesse capítulo e aquelas criadas nos exercícios anteriores, faça um programa para ler um cadeia de caracteres `s` e exibir: (*a*) uma árvore de Huffman correspondente, (*b*) os códigos dos caracteres de `s`, (*c*) a cadeia de dígitos resultante da compressão de `s` e (*d*) a taxa de compressão obtida.