Estruturas de Dados

Aula 19 — Compressão de Dados

Prof Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia — Campus Feira de Santana

2026

Agenda

O que é Compressão de Dados?

Algoritmo de Huffman

Aplicações

Exercícios

O que é Compressão de Dados?

Representar a mesma informação com menos bits

Motivação: o problema

Cenário

Um texto usa um alfabeto de símbolos com **frequências desiguais**. A codificação padrão (ex.: ASCII, 8 bits/caractere) gasta o mesmo número de bits para símbolos raros e frequentes.

Ideia central

Atribuir códigos **mais curtos** a símbolos **mais frequentes** e códigos **mais longos** a símbolos raros – reduzindo o tamanho médio em bits.

Codificação de tamanho fixo

Com 5 símbolos distintos, um código de tamanho fixo precisa de $\lceil \log_2 5 \rceil = 3$ bits por símbolo – mesmo que um deles apareça 50% das vezes.

Origem

Proposta por David A. Huffman em 1952, ainda estudante de doutorado no MIT, como resposta a um trabalho da disciplina de teoria da informação.

Códigos de Prefixo (Prefix Codes)

Definição

Um código de prefixo é um esquema de codificação em que **nenhum** código é prefixo de outro código. Isso garante que a sequência de bits pode ser decodificada **sem ambiguidade**, sem precisar de separadores.

Sem essa propriedade...

Se `A = 0` e `B = 01`, a sequência `01` pode significar `AB` ou apenas `B` – ambíguo!

Árvore binária → código

Qualquer **árvore binária** cujas folhas são os símbolos define naturalmente um código de prefixo: o código de um símbolo é o caminho raiz→folha (0 = esquerda, 1 = direita).

Vantagem

Decodificação simples: percorrer a árvore bit a bit até atingir uma folha, emitir o símbolo, voltar à raiz.

Algoritmo de Huffman

Construção via fila de prioridade

Construção da Árvore: Ideia

Algoritmo guloso

1. Criar um nó-folha para cada símbolo, com peso = sua frequência.
2. Inserir todos os nós em uma **fila de prioridade (min-heap)**, ordenada por peso.
3. Repetir até restar um único nó:
   1. Remover os **dois** nós de menor peso.
   2. Criar um novo nó interno, com esses dois como filhos e peso = soma dos pesos.
   3. Inserir o novo nó na fila.
4. O nó restante é a **raiz** da árvore de Huffman.

Por que sempre os dois menores?

Combinar os dois símbolos menos frequentes garante que eles fiquem nas posições **mais profundas** da árvore (códigos mais longos), enquanto os mais frequentes ficam perto da raiz (códigos mais curtos).

Complexidade

Com n símbolos: cada remoção/inserção no heap custa $\mathcal{O}(\log n)$, repetida $n-1$ vezes $\Rightarrow$ $\mathcal{O}(n \log n)$ no total.

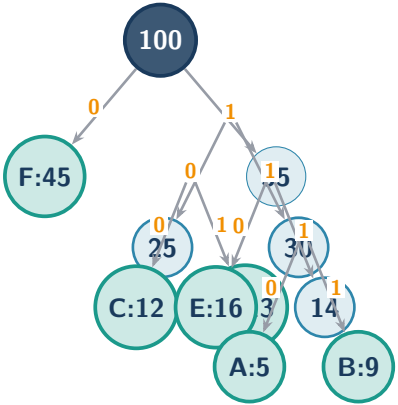
Construção da Árvore: Exemplo

Frequências

Símbolo	Freq.
A	5
B	9
C	12
D	13
E	16
F	45

Exemplo clássico (Cormen et al.) com 6 símbolos e total de 100 ocorrências.

Árvore de Huffman resultante



Estrutura do Nó

Campos do nó

Cada nó guarda o símbolo (apenas em folhas), o peso (frequência acumulada) e ponteiros para os dois filhos.

Folha vs. nó interno

Uma **folha** representa um símbolo original; um **nó interno** (criado durante a fusão) tem peso = soma dos pesos dos filhos e símbolo indefinido (`'\0'`).

```c
typedef struct HNo {
  char simbolo;//válido se é folha
  int peso;
  struct HNo *esq, *dir;
} HNo;

HNo* criarFolha(char simbolo,int peso){
  HNo *no =(HNo*)malloc(sizeof(HNo));
  no->simbolo = simbolo;
  no->peso = peso;
  no->esq = no->dir = NULL;
  return no;
}
```

Fila de prioridade

Implementada como **min-heap**: o nó de menor peso fica sempre acessível em $\mathcal{O}(1)$, e inserir/remover custa $\mathcal{O}(\log n)$.

Criando um nó interno

Ao combinar dois nós (os de menor peso da fila), cria-se um novo nó interno cujo peso é a soma dos pesos dos dois filhos.

```
1 HNo* criarInterno(HNo *esq, HNo *dir){
2   HNo *no = (HNo*) malloc(sizeof(HNo));
3   no->simbolo = '\0';
4   no->peso = esq->peso + dir->peso;
5   no->esq = esq;
6   no->dir = dir;
7   return no;
8 }
```

Implementação: Construção da Árvore

Passo a passo

Usando um min-heap (`MinHeap`) já populado com uma folha por símbolo: enquanto houver mais de um nó na fila, remover os dois de menor peso e inserir o nó interno combinado.

Caso especial

Se houver apenas **um símbolo** distinto, é preciso tratar separadamente (a árvore teria só uma folha, sem código bem definido).

```
1 HNo* construirHuffman(MinHeap *fila){
2   while (heapTamanho(fila) > 1) {
3     HNo *esq = heapRemoverMin(fila);
4     HNo *dir = heapRemoverMin(fila);
5     HNo *interno=criarInterno(esq,dir);
6     heapInserir(fila, interno);
7   }
8   return heapRemoverMin(fila);//raiz
9 }
```

Geração dos Códigos

Percurso da árvore

Após construída a árvore, percorrer recursivamente acumulando os bits do caminho: 0 ao ir para a esquerda, 1 ao ir para a direita. Ao atingir uma **folha**, o código acumulado é o do símbolo correspondente.

Complexidade

$\mathcal{O}(n)$ nós visitados, gerando n códigos – cada código tem, no máximo, $\mathcal{O}(n)$ bits no pior caso (árvore degenerada).

```
1  void gerarCodigos(HNo *no, char *codigo, int
       profundidade) {
2    if(no->esq==NULL && no->dir==NULL){
3      codigo[profundidade] = '\0';
4      printf("%c: %s\n", no->simbolo, codigo);
5      return;
6    }
7    codigo[profundidade] = '0';
8    gerarCodigos(no->esq,codigo,profundidade+1);
9    codigo[profundidade] = '1';
10   gerarCodigos(no->dir,codigo,profundidade+1);
11 }
```

Aplicações

Compactação de arquivos e texto

Formatos que usam Huffman

- **DEFLATE** (usado em ZIP, gzip, PNG): combina LZ77 com Huffman para comprimir os resultados.
- **JPEG**: usa Huffman na etapa final, depois da transformada DCT e quantização.
- **MP3 e outros codecs**: aplicam codificação de entropia semelhante à dos coeficientes já processados.

Taxa de compressão

O ganho depende de quão **desigual** é a distribuição de frequências: textos com poucos caracteres muito frequentes comprimem melhor do que dados aleatórios ou uniformes já existentes.

Limite teórico

Pelo **teorema da fonte de Shannon**, a entropia H da fonte é o limite inferior do número médio de bits por símbolo; Huffman se aproxima desse limite (é ótimo entre códigos de prefixo, mas não atinge H exatamente quando as probabilidades não são potências de $1/2$).

Huffman Estático vs. Adaptativo

Huffman estático

Lê todo o arquivo, conta as frequências, constrói **uma** árvore e a usa para todo o arquivo. A árvore (ou tabela de códigos) precisa ser **armazenada** junto com os dados comprimidos, para a decodificação.

Huffman adaptativo

Atualiza a árvore **durante** a codificação, conforme novos símbolos são lidos – não precisa transmitir a árvore, mas é mais complexo de implementar (ex.: algoritmo de Vitter).

Comparação com outras técnicas de compressão

Huffman comprime por **frequência de símbolos individuais** (codificação de entropia). Técnicas como **LZ77/LZ78** exploram **repetições de trechos** (substrings) – por isso os formatos modernos (DEFLATE, LZMA) combinam ambas as ideias.

Exercícios

Pratique Huffman

Exercícios

1. **Construção manual:** Dadas as frequências A:2, B:3, C:4, D:5, E:9, construa a árvore de Huffman passo a passo (mostrando o estado da fila de prioridade a cada iteração) e escreva o código binário de cada símbolo.

2. **Tamanho comprimido:** Usando os códigos do exercício anterior, calcule o número total de bits necessários para codificar a string `"ABCDE"` repetida conforme as frequências acima, e compare com a codificação de tamanho fixo (3 bits/símbolo).

3. **Implementação – decodificarHuffman:** Implemente uma função que recebe a raiz da árvore de Huffman e uma sequência de bits (string de `'0's` e `'1's`) e retorna a string decodificada.

4. **Análise:** Explique por que a árvore de Huffman, embora ótima entre códigos de prefixo, não é necessariamente única: dê um exemplo de frequências em que existam duas árvores de Huffman válidas e igualmente ótimas.

Fim da Aula 19

Dúvidas e próximos passos

Encerrando a Aula 19

Resumo da aula

- **Compressão**: símbolos frequentes ganham códigos mais curtos
- **Código de prefixo**: decodificação sem ambiguidade
- **Huffman**: algoritmo guloso, construção via min-heap, $\mathcal{O}(n \log n)$
- **Ótimo** entre códigos de prefixo (não necessariamente único)
- **Aplicações**: ZIP/gzip/PNG (DEFLATE), JPEG, MP3
- **Estático vs. adaptativo**; combinação com LZ77/LZ78 nos formatos modernos

carolsoko@ifba.edu.br

Dúvidas? Mande uma mensagem!