Estudo de caso: codificação de Huffman (parte II)

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

24 de julho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

Crédito de parte das imagens, a menos se especificado: Wikipedia

Objetivos

- Aprender a efetuar E/S orientada a bits
- Compreender a decodificação de Huffman

Codificação binária

Codificação binária

E/S orientada a bits

- Até o momento, nosso codificador gera “binário falso”
- Para gerar “binário real”, é necessário escrever bits individuais numa porta (i.e. E/S orientada a bits)
- Porém, isso não é possível nativamente (i.e. conseguimos escrever, no mínimo, 1 byte)
- Para efetuar E/S orientada a bits, podemos aplicar a técnica de *bit buffer*:
  1. Acumulamos até 8 bits em um buffer de 1 byte
  2. Quando o buffer encher, convertemos para byte e o escrevemos na porta
- Vamos reunir essa funcionalidade em uma biblioteca

Biblioteca `bin-io.rkt`

E/S orientada a bits

- Note que o buffer é representado por uma lista persistente

```
(define (make-empty-buff)
  null)

(define (buff-occ buff)
  (length buff))

(define (buff-full? buff)
  (= 8 (buff-occ buff)))

(define (buff-empty? buff)
  (null? buff))

(define (buff-write-bit bit buff)
  (let ([buff-a (cons bit buff)])
    (if (buff-full? buff-a)
        (buff-flush buff-a)
        buff-a)))

(define (buff->byte buff)
  (let loop ([pow 128]
             [byte 0]
             [buff (reverse buff)])
    (cond [(null? buff) byte]
          [(= 1 (car buff))
           (loop (/ pow 2)
                 (+ byte pow)
                 (cdr buff))]
          [(= 0 (car buff))
           (loop (/ pow 2)
                 byte
                 (cdr buff))])))

(define (byte->buff byte)
  (let loop ([i 0] [buff null] [byte byte])
    (if (= i 8)
        buff
        (loop (add1 i)
              (cons (remainder byte 2) buff)
              (quotient byte 2)))))

(define (buff-flush buff)
  (if (buff-empty? buff)
      buff
      (begin
        (write-byte (buff->byte buff))
        (make-empty-buff))))
```

Codificador com E/S orientada a bits

Adaptação

- Para adaptar o codificador, definimos:

```
(define (write-binary-output huff-code)
    (define (write-binary-code code buff)
      (foldl buff-write-bit buff code))
    (buff-flush
      (foldl write-binary-code (make-empty-buff) huff-code)))
```

- E substituímos write-output por write-binary-output:

```
(with-output-to-file
    out-file
    #:exists 'replace
    (lambda () (write-binary-output huff-code)))
```

Codificador com E/S orientada a bits

Teste

```
$ racket huffman-encoder.rkt data/marm02.utf8.txt
   data/marm02.utf8.txt.huff
Entropy:  4.546558174934118
Space savings:  0.4277098662488645 3 (max: 1.0)
```

```
ls -l data/
208073 marm02.utf8.txt
119079 marm02.utf8.txt.huff
2467   marm02.utf8.txt.huff.tree
```

- Note que o arquivo tem 119079 bytes = 119079 * 8 bits = 952632 bits
- O valor é aproximado ao aferido anteriormente (952625)
- A diferença de 7 bits correspondente ao padding do último byte (que será no máximo 7)
  - O padding é necessário pois nossa codificação pode gerar comprimentos que não são múltiplos de 8
  - A escolha entre padding de 0s ou 1s é arbitrária

Codificação orientada a bits

Lidando com padding: exemplo

- A codificação orientada a bits exige cuidado devido ao padding no último byte
- Considere de novo o exemplo: w = ABRACADABRA

s	C	D	B	R	A
$C(s)$	100	101	110	111	0

w_c = 0 110 111 0 100 0 101 0 110 111 0

01101110	10001010	11011100

- O código ocupará três bytes, porém no último só ocupa 7 bits
- Se tentássemos decodificar os três bytes, obteríamos: ABRACADABRAA
- Como saber quando o código acaba?

Codificação orientada a bits

Lidando com padding: exemplo

- Para evitarmos erros na decodificação do último símbolo, precisamos marcar o fim do stream (end of stream ou EOS)
- Há duas alternativas comuns:
  1. Contar a quantidade de bits no código e registrar a quantidade num header
  2. Definir um símbolo especial de término e codificar o símbolo
- Vamos optar pela solução 2, pois é mais simples

Codificação orientada a bits

Registrando o símbolo de EOS

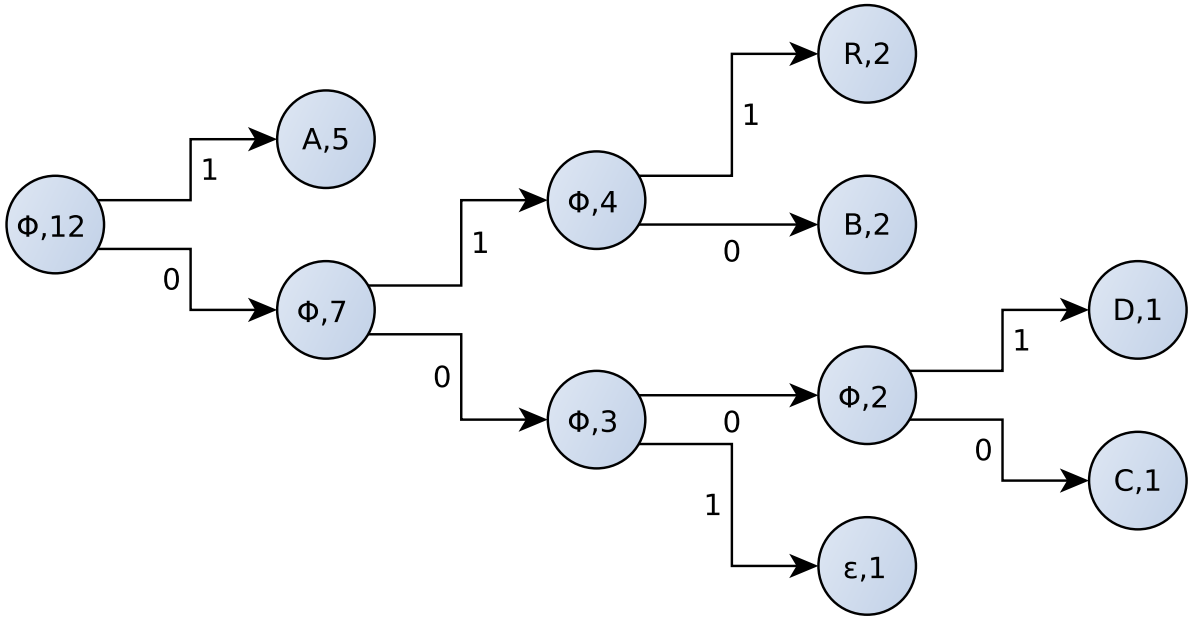
- Como nosso modelo de codificação é baseado nos bytes, o alfabeto é composto pelos $s \in [0, 255]$
- Podemos definir que EOS = 256
- Basta concatenar EOS na entrada e o algoritmo se encarregará de codificá-lo como um símbolo regular:

```
1  (define EOS 256)
2
3  (define (main args)
4    (...)
5
6    (define (append-eos in-data)
7      (append in-data (list EOS)))
8
9    (define input-data
10     (call-with-input-file in-file
11       (lambda (in-port)
12         (append-eos (bytes->list (port->bytes in-port))))))
13
14   (void))
```

Codificação orientada a bits

Registrando o símbolo de EOS

- Considere que EOS=ϵ
- No exemplo w = ABRACADABRAϵ:



s	C	D	B	R	A	ϵ
$f(s)$	1	1	2	2	5	1
$C(S)$	0000	0001	010	011	1	001

w_c = 1 010 011 1 0000 1 0001 1 010 011 1 001

Decodificação de Huffman

Decodificação de Huffman

Métodos de decodificação

- Para decodificar, precisamos enviar o modelo dos dados, pois não é possível derivá-lo a partir da informação codificada
- Como representação do modelo, temos as seguintes alternativas:
  - Histograma (distribuição das frequências)
  - Árvore de Huffman
  - Dicionário de códigos
- A escolha ideal é a que envolve menos bits
- Por simplicidade, vamos:
  - Compartilhar a árvore com o decodificador, usando o arquivo `.tree`
  - Decodificar usando a árvore

Decodificação de Huffman

Métodos de decodificação

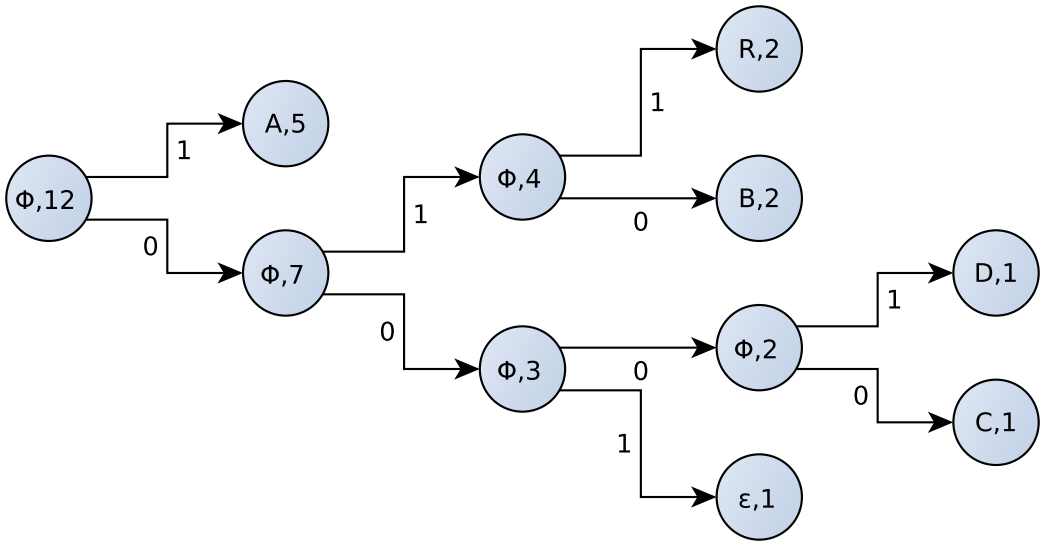
- Para decodificar usando a árvore, percorremos o stream de bits paralelamente à árvore, gerando um símbolo para cada nó-folha encontrado:

Algoritmo de decodificação de Huffman

1. Iniciar na raíz da árvore
2. Para cada bit lido a partir da entrada
   - Se o filho for folha, e o símbolo é EOS, terminar
   - Percorrer o filho da esquerda ou da direita, se o bit for 0 ou 1, respectivamente
   - Se o filho for folha, gerar o símbolo correspondente na saída e reiniciar o percurso na raiz da árvore

Decodificação de Huffman

Exemplo de decodificação



1010011100001000110100111001	A
1010011100001000110100111001	B
1010011100001000110100111001	R
1010011100001000110100111001	A
1010011100001000110100111001	C
1010011100001000110100111001	A
1010011100001000110100111001	D
1010011100001000110100111001	A
1010011100001000110100111001	B
1010011100001000110100111001	R
1010011100001000110100111001	A
1010011100001000110100111001	EOS

Decodificação de Huffman

E/S e coordenação do processo

- Assume-se que o arquivo `.tree` esteja no mesmo diretório que o arquivo de entrada

```
1  (define (main args)
2    (cond [(< (vector-length args) 2)
3           (display "Usage: huffman-decoder <coded-file> <decoded-file>\n" (current-error-port))
4           (exit 1)])
5
6    (define (write-output out-data)
7      (map (lambda (x) (write-byte x)) out-data))
8
9    (define (read-binary-input in-port)
10     (define (parse-byte byte lst) (foldr cons lst (byte->buff byte)))
11     (foldr parse-byte null (bytes->list (port->bytes in-port))))
12
13   (define in-file (vector-ref args 0))
14   (define out-file (vector-ref args 1))
15   (define tree-file (string-append (vector-ref args 0) ".tree"))
16
17   (define in-data
18     (call-with-input-file in-file
19       (lambda (in-port)
20         (read-binary-input in-port))))
21
22   (define huff-tree
23     (call-with-input-file tree-file read))
24
25   (define out-data (huff-decode in-data huff-tree))
26
27   (with-output-to-file
28     out-file
29     #:exists 'replace
30     (lambda () (write-output out-data)))
```

Decodificação de Huffman

Procedimento de decodificação

- Decodificação:

```
1 (define (huff-decode input-data huff-tree)
2   (let loop ([in input-data] [tree huff-tree] [out null])
3     (cond [(null? in) (error "Premature EOS")]
4           [(eq? (node-symbol tree) EOS) (reverse out)]
5           [(node-leaf? tree)
6            (loop in huff-tree (cons (node-symbol tree) out))]
7           [(= 0 (car in)) (loop (cdr in) (node-left tree) out)]
8           [(= 1 (car in)) (loop (cdr in) (node-right tree) out)])))
```

Decodificação de Huffman

Testes

```
$ racket huffman-encoder.rkt data/marm02.utf8.txt
  data/marm02.utf8.txt.huff
Entropy:  4.546628163835074
Space savings:  0.427701202456818 2 (max: 1.0)

$ racket huffman-decoder.rkt data/marm02.utf8.txt.huff
    data/marm02.utf8.txt.huff.decoded

$ ls -l data/
total 536
208073 marm02.utf8.txt
119081 marm02.utf8.txt.huff
208073 marm02.utf8.txt.huff.decoded
2488   marm02.utf8.txt.huff.tree

$ diff data/marm02.utf8.txt data/marm02.utf8.txt.huff.decoded
$
```

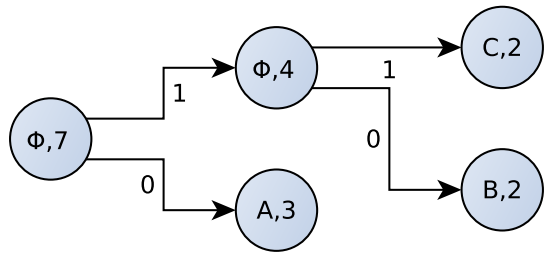
- O arquivo decodificado é idêntico ao original, portanto o decodificador está operando corretamente

Exercícios

Exercício 1

Atualmente, codificador e decodificador compartilham a árvore de Huffman via um arquivo `.tree`, que nada mais é do que a representação da árvore como S-expression.

Um modo mais genérico de codificar a árvore de Huffman consiste em: percorrer a árvore em pré-ordem, gerando 0 para aresta esquerda e 1 para aresta direita. Quando atingirmos um nó-folha, emitimos o código binário do símbolo.



Considerando que os códigos ASCII binários para A, B e C são, respectivamente, 1000001, 1000010 e 1000011, a codificação pré-ordem da árvore da figura é: 0100000110100001011000011.

1. Construa o programa que codifique a árvore de Huffman segundo o processo acima. A chamada dever ser assim:

```
./huff-tree-encoder <tree s-expr> <tree bin>
```

2. Construa o programa que decodifique a árvore binária de Huffman:

```
./huff-tree-decoder <tree bin> <tree s-expr>
```

Estudo de caso: codificação de Huffman (parte II)

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

24 de julho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

Crédito de parte das imagens, a menos se especificado: Wikipedia