Estudo de caso: codificação de Huffman (parte I)

Com fundamentos de E/S em Racket

Profs. Diogo S. Martins e Emilio Francesquini

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

MCTA016 - Paradigmas de Programação (Prática)

17 de julho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

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

Objetivos

- Apresentar as principais APIs para efetuar E/S em Racket
- Praticar os temas vistos até o momento com uma aplicação de compressão de texto

Entrada e saída em Racket

Portas

Porta (em Racket)

Mecanismo sequencial para leitura e escrita de dados.

- **Porta de entrada**: mecanismo de leitura de dados
- **Porta de saída**: mecanismo de escrita de dados
- Modalidades:
  - **Arquivos**
  - **Strings**
  - **Conexões TCP**
  - **Processos (E/S padrão e pipe)**
  - **Pipes internos**

Atenção

E/S gera efeito colateral, portanto uma função que efetua E/S automaticamente é referencialmente opaca (i.e. não é pura). É considerado boa prática modularizar código que efetua E/S separadamente do código funcional (isso vale, inclusive, para programação imperativa procedural).

Mais detalhes: `https://docs.racket-lang.org/guide/ports.html`

Procedimentos de escrita

- `print`: formatação análoga à do REPL
- `write`: formatação compatível com `read`
- `display`: reduz caracteres e strings ao seu conteúdo
- outros: `printf`, `println`, `writeln`, `write-bytes`, etc.
- Referência: `https://docs.racket-lang.org/reference/Writing.html`

```
> (print 1/2)
1/2
> (print #\x)
#\x
> (print "hello")
"hello"
> (print #"goodbye")
#"goodbye"
> (print '|pea pod|)
'|pea pod|
> (print '("i" pod))
'("i" pod)
> (print write)
#<procedure:write>
```

```
> (write 1/2)
1/2
> (write #\x)
#\x
> (write "hello")
"hello"
> (write #"goodbye")
#"goodbye"
> (write '|pea pod|)
|pea pod|
> (write '("i" pod))
("i" pod)
> (write write)
#<procedure:write>
```

```
> (display 1/2)
1/2
> (display #\x)
x
> (display "hello")
hello
> (display #"goodbye")
goodbye
> (display '|pea pod|)
pea pod
> (display '("i" pod))
(i pod)
> (display write)
#<procedure:write>
```

Saída padrão

- Todos os procedimentos de escrita usam uma porta: quando omitida, usa-se implicitamente a saída padrão
- Saída padrão é acessada via `(current-outpot-port)`:

```
> (write "hello" (current-output-port))
"hello"
> (write "hello")
"hello"
```

- Podemos redirecionar a saída padrão para outra porta com os procedimentos `with-output-to-*`:

```
> (with-output-to-file "test.txt"
    (lambda () (write "hello file")))
> (with-output-to-string
    (lambda () (write "hello string")))
"\"hello string\""
```

- Os procedimentos `with-output-to-*` tem uma vantagem: abrem e fecham a porta implicitamente, i.e. abre no início e fecha no final

Note que os procedimentos `with-output-to-*` recebem uma *thunk*, i.e. uma função anônima sem argumentos. Thunks são muito usadas em programação funcional para injetar comportamentos — nesse caso, é necessária pois o corpo só poderá ser executado quando a saída padrão estiver pronta, i.e. devidamente redirecionada.

Portas de arquivos

- Criação de arquivos explicitamente demanda a criação de uma porta de saída:

```
> (define a-file-port
   (open-output-file "hello.txt"
                     #:exists 'replace))
> (write "hello again" a-file-port)
> (close-output-port  a-file-port)
> ,shell cat hello.txt
"hello again"
```

  Portas abertas explicitamente precisam ser fechadas explicitamente, pois consomem recursos do SO

- Idem para porta de entrada:

```
> (define a-in-port (open-input-file "hello.txt"))
> (read a-in-port)
"hello again"
> (close-input-port a-in-port)
```

  Podemos usar também `with-input-from*` (recomendado):

```
> (with-input-from-file "hello.txt" (lambda () (read)))
"hello again"
```

Leitura e escrita de dados em arquivos

- Podemos ler e escrever estruturas Racket em arquivos:

```
> (define (gen-random-list n)
    (map (lambda (x) (random)) (make-list n 0)))
> (with-output-to-file "random-list.dat"
    (lambda () (write (gen-random-list 15))))
> ,shell cat "random-list.dat"
(0.6818480724059975 0.2934496770700284 0.8314654969016145
0.3823584491690988 0.07833609038356384 0.5206019713741751
0.004425747767220143 0.5510021589250419 0.7638412313254029
0.7756012688216437 0.5107147354233696 0.5914384978868086
0.18727365856825398 0.8812702787817033 0.4561075067311436)
> (define a-random-list
    (with-input-from-file
      "random-list.dat" read))
> (length a-random-list)
15
```

- Esse recurso é muito útil para trabalhar com a memória secundária e ao mesmo tempo evitar parseamento explícito dos dados
- Podemos, inclusive, carregar programas e executar com `eval`[1]:

```
> (define prog (with-input-from-file "hello.rkt" (lambda () (read))))
> prog
'(let () (print "Hello, i am a Racket program"))
> (eval prog)
"Hello, i am a Racket program"
```

[1] Isso é possível devido à homoiconicidade de Lisp, como vimos em teoria.

E/S com chars vs. E/S com bytes

- Operam com char: `read-line`, `read`, `display`, `write`, etc.
- Operam com bytes: `read-byte`, `write-byte`, etc.

```
> (with-input-from-file "random-list.dat"
    (lambda () (read-byte)))
40
> (integer->char (with-input-from-file "random-list.dat"
    (lambda () (read-byte))))
#\(
> (with-input-from-file "random-list.dat"
    (lambda () (read-bytes 10)))
#"(0.6818480"
```

- Note que que o resultado de (`read-bytes` n) é uma byte string de comprimento n, que comporta-se como um `vector` de bytes *mutável*
  Portanto, para processá-lo como lista, precisamos converter:

```
> (bytes->list
    (with-input-from-file "random-list.dat"
      (lambda () (read-bytes 10))))
'(40 48 46 54 56 49 56 52 56 48)
```

- Já temos as noções básicas, para mais informações, consultar:
  `https://docs.racket-lang.org/reference/input-and-output.html`

Exercícios de E/S

Exercício 1

A fatoração de um inteiro em primos, se feita de modo ingênuo, costuma ser custosa pois os algoritmos básicos de geração de números primos não são eficientes. Uma forma de agilizar o processso consiste em utilizar uma lista de números primos pré-computados.

Construa três programas:

1-a. `gen-primes.rkt`: que gera todos os números primos até um valor `n`. O programa deve ser chamado assim (supondo que `n=1000` e estou salvando para o arquivo `primes.csv`):

```
$ ./gen-primes.rkt 1000 primes.csv
```

1-b. `factorize.rkt`: usa uma lista pré-computada de números primos para fatorar um inteiro. Deve ser chamado assim (`2783` é o número a fatorar e `factors.csv` é a lista de fatores):

```
$ ./factorize.rkt 2783 primes.csv factors.csv
```

1-c. `unique.rkt`: recebe um arquivo de fatores e elimina os fatores duplicados (e.g. se os fatores de um número são `2 2 3 3 4`, mantém só `2 3 4`). Restrição: o algoritmo de eliminar duplicidade só percorrer uma única vez a lista de fatores. Exemplo de chamada (`unique-factors.csv` é o arquivo de saída do filtro):

```
$ ./unique.rkt factors.csv unique-factors.csv
```

Exercício 2

2-a. Construa o programa `random-sorted-lists.rkt <n> <k> <m>` que gere n listas de inteiros aleatórios, no intervalo $[0, k)$, cada lista contendo m elementos. Cada lista deve ser salva em um arquivo separado, com nomes sequenciais, no diretório atual. Por exemplo:

```
$ ./random-sorted-lists.rkt 3 100 10
$ ls
1.csv 2.csv 3.csv
$ cat 1.csv
8 26 27 38 40 50 53 55 69 93
```

2-b. Construa o programa `merge.rkt <dir> <res>` que combine todas as listas ordenadas do diretório <dir> em um única lista no arquivo <res>, a qual também deve estar ordenada. Por exemplo:

```
$ ls data/
1.csv 2.csv
$ cat data/1.csv
46 90 95
$ cat data/2.csv
3 85 92
$ ./merge.rkt data data/res.csv
$ ls data/
1.csv 2.csv res.csv
$ cat res.csv
3 46 85 90 92 95
```

Estudo de caso: codificação de Huffman

Codificação de Huffman

Revisão

Código binário de comprimento variável

Código no qual diferentes símbolos são representados com quantidades variáveis de bits, em geral dependendo de um modelo de probabilidades.

- ASCII é código de comprimento fixo (i.e. todo símbolo tem 8 bits, sendo 7 de informação + 1 de paridade)
- Se tivermos uma distribuição de probabilidade dos símbolos, podemos atribuir:
  - Códigos mais curtos para símbolos mais frequentes
  - Códigos mais longos para símbolos menos frequentes
  - Uma mensagem (e.g. usando um alfabeto de caracteres) assim codificada pode apresentar compressão

Código livre de prefixo

Código em que cada palavra não é prefixo de outra palavra. Também denominado código auto-pontuável.

Codificação de Huffman

Revisão

Código livre de prefixo ótimo

Código livre de prefixo em que toda palavra tem a menor largura possível

Árvore de um código binário ótimo

A árvore binária correspondente a um código binário ótimo tem os nós-folha com as menores alturas possíveis.

Codificação de Huffman

Algoritmo que constrói um código livre de prefixo ótimo.

- Intuição: construção da árvore binária do código com atenção à minimização das alturas, com base na distribuição das probabilidades dos símbolos

Codificação de Huffman

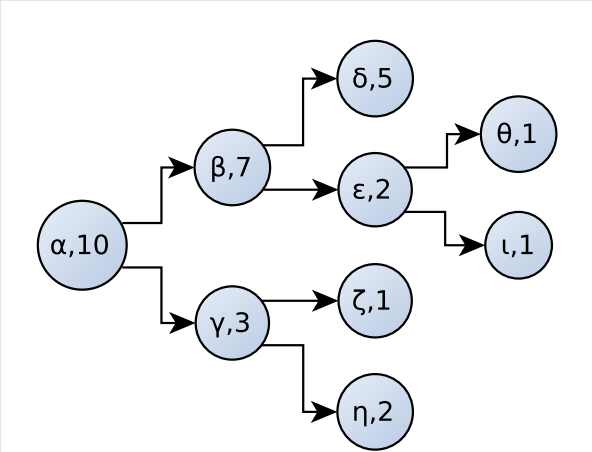
Revisão

Árvore binária com frequências

Uma árvore binária com frequências tem as seguintes propriedades:

- Cada nó é rotulado por um par (s, f), em que s é um símbolo e f é a frequência do símbolo
- A frequência armazenada em um nó não-folha é a soma das frequências armazenadas nos nós filho

Exemplo 1: Árvores com frequências



- $\alpha = \beta + \gamma$
- $\beta = \delta + \epsilon$
- $\epsilon = \theta + \iota$
- $\ldots$

Codificação de Huffman

- Intuição:
  - Construir uma árvore binária com frequências[2] para o código
  - A árvore é construída começando pelas folhas (o último nó criado é a raiz)
  - A construção ocorre em ordem crescente de frequências
  - Utilizar a árvore para codificar a mensagem

Algoritmo de construção da árvore de Huffman

1. Criar um nó folha com frequência para cada símbolo (construir uma "floresta" de altura 0)
2. Enfileirar os nós em ordem crescente de frequência
3. Enquanto houver mais de um nó na fila:
   1. Remover os dois nós/árvores com as menores frequências
   2. Criar um novo nó pai tendo os dois nós como filhos (criar uma nova árvore)
   3. Inserir a nova árvore na fila (respeitando a ordem crescente)
4. A última árvore restante na fila é a árvore de Huffman

[2] Aqui utilizamos frequências, que são proporcionais às probabilidades

Codificação de Huffman

Exemplo

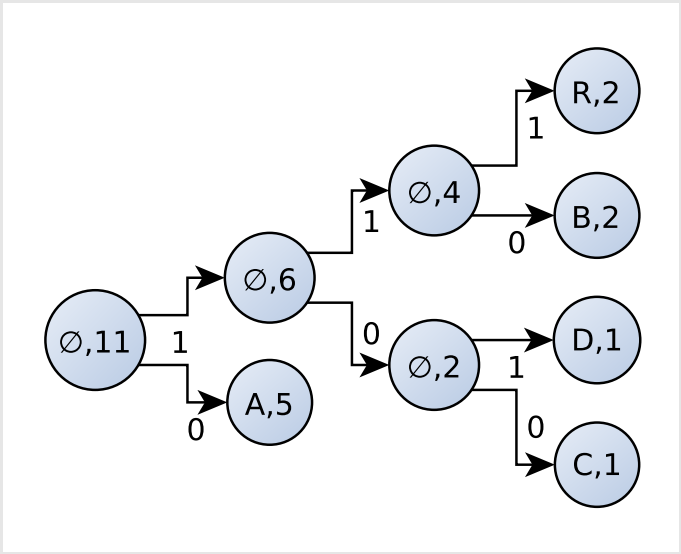
Exemplo 2: codificação de Huffman (parte I)

Codifique a palavra em código de Huffman.

ABRACADABRA

- Construção da árvore de Huffman

s	f
C	1
D	1
B	2
R	2
A	5



Codificação de Huffman

Exemplo

Exemplo 2: codificação de Huffman (parte II)

Codifique a palavra em código de Huffman.

ABRACADABRA

- Construção do dicionário (cada código é o percurso da raiz ao nó-folha)
- Codificação é a tradução de cada símbolo para seu respectivo código

s	**cód.**
A	0
B	110
C	100
D	101
R	111

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

Codificação de Huffman

Exemplo

Exemplo 2: codificação de Huffman (parte III)

- Análise da economia de espaço
  $|w| = 11 \cdot 8$ bits = 88 bits
  $|w_c| = 23$ bits
  $S_c = 1 - \frac{23}{88} = 0.7386$
- S_c não está considerando o custo de enviar o dicionário ou a árvore junto com a mensagem
- Ainda não sabemos como codificar em binário o dicionário ou a árvore
- Na prática, o dicionário ou a árvore aumentarão a quantidade de bits de w_c e, consequentemente, diminuirão a economia de espaço
- Para simplificar, vamos assumir que o envio da árvore traz um impacto $\Delta < 0$ na economia de espaço
- Então: $S_c = 0.7386 + \Delta$

Demonstração: algoritmo de **codificação** de Huffman

1. Modelagem dos dados
2. Modelagem de uma fila de prioridade mínima
3. Modelagem da árvore de Huffman
4. Entrada de dados
5. Modelagem do dicionário de Huffman
6. Algoritmo de codificação
7. Saída de dados
8. Testes

Modelagem dos dados

- Huffman pode ser aplicado em diferentes modalidades de dados
- Basta definir qual será o alfabeto (i.e. os símbolos)
- Para texto, podemos definir que cada símbolo é um caractere
- Para comprimir textos independentemente da codificação de caracteres, podemos assumir que um byte é um símbolo
- Em suma, o nosso alfabeto serão os diferentes bytes presentes num arquivo de texto[3]

[3] Nota: estamos fazendo uma simplificação pois, a depender da codificação, um caracter pode corresponder a mais de um byte (e.g. em UTF-8, um caracter pode ter até 4 bytes)

Modelagem de uma fila de prioridade mínima

- Racket possui uma implementação de min-heap, porém é imperativa
- Podemos implementar nossa própria fila de prioridade puramente funcional
- Para simplificar, podemos adotar uma implementação com listas ordenadas e funções de ordem superior (conforme caiu na P1):

```
1  (define (pq-enqueue v pq pred)
2    (cond [(null? pq) (list v)]
3          [(pred v (car pq)) (cons v pq)]
4          [else (cons (car pq)
5                      (pq-enqueue v (cdr pq) pred))]))
6
7  (define (pq-dequeue pq)
8    (cond [(null? pq) null]
9          [else (cdr pq)]))
10
11 (define (pq-peek pq)
12   (if (null? pq)
13       pq
14       (car pq)))
15
16 (define (pq-empty? pq)
17   (null? pq))
18
19 (define (pq-has-one? pq)
20   (cond [(null? pq) #f]
21         [(null? (cdr pq)) #t]
22         [else #f]))
```

- Nota: essa PQ não é a mais eficiente possível, pois `pq-enqueue` é $O(n)$
- A versão com enfileiramento $O(log(n))$ é mais trabalhosa, além de não ser o foco do problema nesse momento

Modelagem da árvore de Huffman

- Podemos modelar os nós da árvore como listas:

```
;;;;;;;;;;;; Huffman tree API

(define (make-huff-node symbol freq left right)
  (list symbol freq left right))

(define (node-symbol node)
  (first node))

(define (node-freq node)
  (second node))

(define (node-left node)
  (third node))

(define (node-right node)
  (fourth node))

(define (node-leaf? node)
  (and (null? (node-left node)) (null? (node-right node))))
```

Modelagem da árvore de Huffman

- Para construir nós internos definimos:

```
1 ; build an internal huff node
2 (define (merge-huff-nodes node-left node-right)
3   (make-huff-node
4     null
5     (+ (node-freq node-left) (node-freq node-right))
6     node-left
7     node-right))
```

- Para facilitar a operações dos nós na fila de prioridades definimos um wrapper:

```
1 ; huffman wrapper for the priority queue
2 (define (pq-huff-enqueue v pq)
3   (define (less-than-huff-node node-a node-b)
4     (< (node-freq node-a) (node-freq node-b)))
5   (pq-enqueue v pq less-than-huff-node))
```

Entrada dos dados via arquivo

- Como E/S geram efeitos colaterais, é imprescindível mantê-la fora de nossas funções puras
- Por exemplo, podemos "isolar" toda E/S em um algum tipo módulo "impuro"
- Uma possibilidade é concentrar toda E/S em um procedimento `main`:

```
1 (define (main args)
2   ;;;;; processamento com efeitos colaterais
3
4   (void))
5
6 (main (current-command-line-arguments))
```

- `(current-command-line-arguments)` retorna os argumentos do programa como um `vector`
- Nesse exemplo, iniciamos o programa com uma chamada para `main`, passando o argumentos do programa

Para parseamento mais sofisticado de argumentos da linha do comando (i.e. estilo POSIX), usar a biblioteca `racket/cmdline` (urlhttps://docs.racket-lang.org/reference/Command-Line_Parsing.html)

Entrada de dados

- Quero chamar o programa assim:

```
$ racket huffman-encoder.rkt in.txt out.huff
```

- Para processar/validar os argumentos do programa:

```
1 (define (main args)
2   (cond [(< (vector-length args) 2)
3          (display "Usage: huffman-encoder
4                    <uncoded-file> <coded-file>\n"
5                   (current-error-port))
6          (exit 1)])
7          ...)
```

Entrada de dados

- Para carregar um arquivo de entrada, orientado a bytes:

```
1 (define in-file
2     (vector-ref args 0))
3 (define input-data
4     (call-with-input-file in-file
5       (lambda (in-port)
6         (bytes->list (port->bytes in-port)))))
7   (define bytes-in-count
8     (length input-data))
```

- Estamos carregando todo arquivo na memória em uma passada
- Embora mais prático, esse método não é robusto, pois torna-se inviável para arquivos grandes
- Para minimizar o uso de memória primária, o carregamento incremental demandaria a criação de uma repetição na qual[4]:
  - Lê-se um byte (ou bloco de bytes)
  - Para cada byte lido, efetua-se a sua codificação
  - Para cada código criado, escreve-se no arquivo de saída
- A implementação desse processo fica como exercício

[4] Supondo, claro, que houve uma passada intermediária para modelar as probabilidades, ou que o modelo tenha sido pré-definido.

Construção do modelo

- Tendo os dados de entrada, podemos construir o modelo dos dados
- No nosso caso, será um histograma das frequências dos bytes:

```
(define hist
    (build-histogram input-data))
```

```
; count the frequency of each byte, storing
; in a hashtable
(define (build-histogram input-data)
  (define (update-histogram symbol hist)
    (hash-update hist symbol add1 0))
  (foldr update-histogram (hash) input-data))
```

Construção da floresta de Huffman

- Com base no modelo de probabilidade, precisamos construir os futuros nós-folha da árvore de Huffman
- E colocá-los na fila de prioridades:

```
1  ; huffman wrapper for the priority queue
2  (define (pq-huff-enqueue v pq)
3    (define (less-than-huff-node node-a node-b)
4      (< (node-freq node-a) (node-freq node-b)))
5    (pq-enqueue v pq less-than-huff-node))
6
7  ; populate the priority queue
8  (define (build-pq hist)
9    (define (build-single-node k v)
10             (make-huff-node k v null null))
11   (foldr pq-huff-enqueue
12          null
13          (hash-map hist build-single-node)))
```

Construção da árvore de Huffman

- A partir da fila populada, construímos a árvore:

```
1  ; tree-building algorithm
2  (define (build-huff-tree pq)
3    (let loop ([pq pq])
4      (cond [(pq-empty? pq) pq]
5            [(pq-has-one? pq) (pq-peek pq)]
6            [else
7             (let* ([node-a (pq-peek pq)]
8                    [pq-a (pq-dequeue pq)]
9                    [node-b (pq-peek pq-a)]
10                   [pq-b (pq-dequeue pq-a)])
11                   (loop (pq-huff-enqueue
12                           (merge-huff-nodes node-a node-b)
13                           pq-b)))])))
```

Construção do dicionário

- Tendo construído a árvore, construímos o dicionário (representado como uma hashtable) via percurso pré-ordem:

```
1  ; derive a code table by traversing the huffman tree
2  (define (build-code-table huff-tree)
3    (let loop ([code-table (hash)]
4               [huff-tree huff-tree]
5               [code null])
6      (cond [(null? huff-tree) code-table]
7            [(node-leaf? huff-tree)
8             (hash-set code-table (node-symbol huff-tree)
9                                  (reverse code))]
10           [else
11            (let* ([code-table-a (loop code-table
12                                       (node-left huff-tree)
13                                       (cons 0 code))]
14                   [code-table-b (loop code-table-a
15                                       (node-right huff-tree)
16                                       (cons 1 code))])
17              code-table-b)])))
```

Codificação

- Agora basta codificar cada símbolo com base no dicionário:

```
1 ; encode each symbol using the code table
2 (define (huff-encode input-data huff-code-table)
3   (define (encode-symbol symbol)
4     (hash-ref huff-code-table symbol))
5   (map encode-symbol input-data))
```

- Por simplificidade, convencionou-se que o resultado da codificação é uma lista de listas
- Cada lista interna é composta pelos “bits” do código
- Na verdade, estamos usando a notação de “falso binário”, i.e., 0s e 1s codificados como inteiros convencionais

A `main` coordena o processo

```
(define (main args)
  (...)

  ;;;;;;;;;;;; IO and helpers
  (define (write-output huff-code)
    (define (write-code code)
      (map (lambda (x) (write x)) code))
    (map write-code huff-code))

  ;;;;;;;;;;;; start encoding
  (define in-file
    (vector-ref args 0))
  (define out-file
    (vector-ref args 1))
  (define tree-file
    (string-append out-file ".tree"))

  (define input-data
    (call-with-input-file in-file
      (lambda (in-port)
        (bytes->list
          (port->bytes in-port)))))
  (define bytes-in-count
    (length input-data))

  (define hist
    (build-histogram input-data))
  (define huff-tree
    (build-huff-tree (build-pq hist)))
  (define huff-code
    (huff-encode input-data (build-code-table huf
  (define bits-out-count
    (foldl + 0 (map length huff-code)))

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

  (with-output-to-file
    tree-file
    #:exists 'replace
    (lambda () (write huff-tree)))

  (void))
```

- **Importante**: a função `write-output` gera o arquivo de saída codificado em "falso binário", i.e. 0s e 1s em ASCII
- Com isso, não detectamos compressão no tamanho dos arquivos, mas sim na quantidade de "falsos bits" que foram gerados

A `main` coordena o processo

- Podemos decorar a saída padrão do programa com algumas métricas:

```
1  (define (space-savings bits-in bits-out)
2    (- 1.0 (/ bits-out bits-in)))
3
4  (define (entropy hist symbol-count)
5    (define (information prob)
6      (* prob (log prob 2)))
7    (define float-symbol-count
8      (exact->inexact symbol-count))
9    (define probs
10     (hash-map hist (lambda (symb freq) (/ freq float-symbol-count))))
11   (- (foldl + 0 (map information probs))))
12
13 (...)
14
15 (printf "Entropy:\t~v\n" (entropy hist bytes-in-count))
16 (printf "Space savings:\t~v (max: 1.0)\n"
17         (space-savings (* 8 bytes-in-count) bits-out-count))
```

```
$ racket huffman-encoder.rkt data/marm02.utf8.txt
    data/marm02.utf8.txt.huff
Entropy:  4.546558174934118
Space savings:  0.42770986624886453 (max: 1.0)
```

Análise da eficiência

- Vamos analisar a eficiência do algoritmo com o exemplo:

```
$ ls -l data/marm02.utf8.txt*
-rw-rw-r-- 1   208073   data/marm02.utf8.txt
-rw-rw-r-- 1   952625   data/marm02.utf8.txt.huff
```

- O arquivo codificado possui 952625 “bits”
- O arquivo original possui $208073 \cdot 8 = 1664584$ bits
- A entropia é de 4.547 bits por símbolo
- Um código ótimo deve ter o comprimento médio de cada símbolo em $[4.547, 5.547)$ bits
- Isso daria $[208073 \cdot 4.547, 208073 \cdot 5.547) = [946107.931, 1154180.930)$ bits
- Portanto nosso código está dentro do intervalo de otimalidade, bem próximo do limite inferior

Próxima aula: decodificação de Huffman

Estudo de caso: codificação de Huffman (parte I)

Com fundamentos de E/S em Racket

Profs. Diogo S. Martins e Emilio Francesquini

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

MCTA016 - Paradigmas de Programação (Prática)

17 de julho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

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