УНИВЕРСИТЕТ ИТМО

ФГАОУ ВО «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Факультет программной инженерии и компьютерной техники

ПРИКЛАДНАЯ МАТЕМАТИКА

Лабораторная работа №4

Построение кода Хэмминга

Лабушев Тимофей

Группа P3302

Санкт-Петербург

2019

Цель работы

Получить практические навыки кодирования и декодирования помехоустойчивыми кодами Хэмминга.

Задание

1. Реализовать процедуру построения кода Хэмминга с заданным числом информационных символов с кодовыми расстояниями 3 и 4.
2. Реализовать процедуру декодирования кода Хэмминга и исправления ошибок для различных кодовых расстояний. Провести программный контроль выполнения на примере случайных кодовых комбинаций.

Исходный код

Кодирование

```
#lang racket

(require data/bit-vector math/base)

(provide encode-sec encode-secded decode-sec decode-secded)

; === Encoding

; A Single Error Correction code (Hamming distance = 3) can either correct a
; single-bit error or detect two-bit errors, but not both at the same time:
; since errors appear the same, if we perform error correction
; we get an incorrect result for a two-bit error.
(define/contract (encode-sec message)
  (-> bit-vector? bit-vector?)

  (define message-bits (add1 (bit-vector-length message)))
  (define message-len
    (if (power-of-two? (add1 message-bits))
      message-bits
      (+ message-bits (- (expt 2 (exact-ceiling (log message-bits 2))) 1 message-bits))))
  (define check-bits (exact-ceiling (log message-len 2)))
  (define encoded-len (+ check-bits message-len))

  (define encoded (make-bit-vector encoded-len))
  (for/fold ([data-i 0] [check-bit-i 0]) ([i (in-range encoded-len)])
    (cond
      [(and (power-of-two? (add1 i)) (check-bit-i . < . check-bits))
        (values data-i (add1 check-bit-i))]
      [else
        (bit-vector-set! encoded i (bit-vector-ref message data-i #f))
        (values (add1 data-i) check-bit-i)]))

  (for ([i (in-range check-bits)])
    (define check-bit-i (sub1 (expt 2 i)))
    (define parity (data-parity-bit encoded check-bit-i))
    (bit-vector-set! encoded check-bit-i parity))

  encoded)

; A Single Error Correction, Double Error Detection code (Hamming distance = 4)
; can both correct a single-bit error and detect two-bit errors.
; It adds a single additional check bit, computed as the parity of all other bits.
(define/contract (encode-secded message)
  (-> bit-vector? bit-vector?)

  (define secbits (bit-vector->list (encode-sec message)))
  (define total-parity (for/fold ([parity #f]) ([b (in-list secbits)])
    (xor parity b)))
  (list->bit-vector (append secbits (list total-parity))))

; === Decoding

(define/contract (decode-sec encoded)
  (-> bit-vector? (values (or/c 'correct integer?) bit-vector?))

  (define (compute-message-len check-bits message-len)
    (if ((- (expt 2 check-bits) check-bits) . >= . message-len)
      (add1 message-len)
      (compute-message-len (add1 check-bits) (sub1 message-len))))

  (define encoded-len (bit-vector-length encoded))
  (define message-len (compute-message-len 0 encoded-len))
  (define check-bits (- encoded-len message-len))

  (define syndromes (for/list ([i (in-range check-bits)])
    (define check-bit-i (sub1 (expt 2 i)))
    (define actual (bit-vector-ref encoded check-bit-i))
    (define expected (data-parity-bit encoded check-bit-i))
    (xor expected actual)))

  (define syndrome-dec (sub1
    (for/sum ([(s i) (in-indexed syndromes)] #:when s) (expt 2 i))))

  (cond
    [(= -1 syndrome-dec)
      (values 'correct (extract-encoded-message-sec encoded check-bits message-len))]
    [else
      (bit-vector-set! encoded syndrome-dec (not (bit-vector-ref encoded syndrome-dec)))
      (values syndrome-dec (extract-encoded-message-sec encoded check-bits message-len))]))

(define (decode-secded encoded)
  (define total-parity-i (sub1 (bit-vector-length encoded)))
  (define total-parity-actual (bit-vector-ref encoded total-parity-i))
  (define total-parity-expected
    (for/fold ([parity #f]) ([(b i) (in-indexed (in-bit-vector encoded))]
                             #:break (= i total-parity-i))
    (xor parity b)))
  (define-values (status decoded)
    (decode-sec (bit-vector-copy encoded 0 (sub1 (bit-vector-length encoded)))))
  (cond
    [(and (eq? status 'correct) (eq? total-parity-actual total-parity-expected))
      (values 'correct decoded)]
    [(eq? status 'correct) ; error in the parity bit
      (values total-parity-i decoded)]
    [(and (integer? status) (not (eq? total-parity-actual total-parity-expected)))
      (values status decoded)] ; single bit error
    [else 'double-error]))

; === Utils

(define (extract-encoded-message-sec encoded check-bits message-len)
  (define message (make-bit-vector message-len))
  (for/fold ([data-i 0] [check-bit-i 0]) ([i (in-range (bit-vector-length encoded))])
  (cond
    [(and (power-of-two? (add1 i)) (check-bit-i . < . check-bits))
      (values data-i (add1 check-bit-i))]
    [else
      (bit-vector-set! message data-i (bit-vector-ref encoded i))
      (values (add1 data-i) check-bit-i)]))
  message)

(define (data-parity-bit data bit-i)
  (for/fold ([parity #f])
            ([data-i (in-range (add1 bit-i) (bit-vector-length data))]
             #:when (= (add1 bit-i) (bitwise-and (add1 bit-i) (add1 data-i))))
    (xor parity (bit-vector-ref data data-i))))
```

Тестирование

```
#lang racket

(require rackunit rackunit/text-ui data/bit-vector "hamming.rkt")

(define hamming-tests
  (test-suite "hamming.rkt tests"

    (test-case "SEC encode-decode with no bit errors"
      (define cases '("101" "10111" "1001000" "1001101010"))
      (for ([msg (in-list cases)])
        (define encoded (encode-sec (string->bit-vector msg)))
        (define-values (status decoded) (decode-sec encoded))
        (check-equal? status 'correct)
       (define expected (~a msg #:min-width (bit-vector-length decoded) #:right-pad-string "0"))
        (check-equal? (bit-vector->string decoded) expected)))

    (test-case "SEC encode-decode with one bit error"
      (define encoded (encode-sec (string->bit-vector "1011101")))
      (bit-vector-set! encoded 0 #t)
      (define-values (status decoded) (decode-sec encoded))
      (check-equal? status 0)
      (check-equal? (bit-vector->string decoded) "1011101"))

    (test-case "SECDED encode-decode with one bit error"
      (define encoded (encode-secded (string->bit-vector "1011101")))
      (bit-vector-set! encoded 0 #t)
      (define-values (status decoded) (decode-secded encoded))
      (check-equal? status 0)
      (check-equal? (bit-vector->string decoded) "1011101"))
    (test-case "SECDED encode-decode with two bit errors"
      (define encoded (encode-secded (string->bit-vector "1011101")))
      (bit-vector-set! encoded 0 #t)
      (bit-vector-set! encoded 1 #t)
      (check-equal? (decode-secded encoded) 'double-error))))

(run-tests hamming-tests)
```

Выводы

В ходе выполнения лабораторной работы были изучены приципы построения помехоустойчивых кодов Хэмминга для исправления одной ошибки (SEC) и исправления одной с обнаружением двух (SECDED).