Домашно упражнение №2

Петко Борджуков (ФN№ 61322)

Задача 1

За да решим задачата, дефинираме обект *Polynom* с полета *coefficients*, *power* и *truth_table*, които съдържат съответно вектора от коефициенти, степента и вярностната таблица на полинома. Инстанция на така зададения обект може да бъде създадена по един от два начина – чрез метода *Polynom#[]=(*coefficients)* по коефициенти, или чрез метода *Polynom#truth_vector(*vector)* по вярностен вектор.

Вярностна таблица на полином по вектор от коефициенти

Вярностната таблица на полином, създаден чрез подаване на вектор от коефициенти, попълваме чрез заместване. Методът *Polynom#truth_table* връща хеш-таблица от вида *вектор от стойности на аргументите => стойност.*

```
007 > require './polynom.rb'
008 > p = Polynom[1, 0, 0, 1, 1, 0, 0, 0]
009 > p.truth_table.each {|key, value| p "%s    => %s" %[key, value]}
"[0, 0, 0]    => 1"
"[1, 0, 0]    => 1"
"[0, 1, 0]    => 1"
"[0, 0, 1]    => 0"
"[1, 1, 0]    => 0"
"[1, 0, 1]    => 0"
"[0, 1, 1]    => 0"
"[1, 1, 1]    => 1"
```

Коефициенти на полином по вярностен вектор

Векторът от коефициенти на полином, създаден чрез подаване на вярностен вектор, намираме по начина за намиране на полином на Жегалкин на булева функция чрез решаване на система, описан в К. Манев - „Увод в дискретната математика“. Методът *Polynom#coefficients* връща вектор от коефициентите на полинома.

```
011 > require './polynom.rb'
012 > p = Polynom.truth_vector(1, 1, 1, 0, 0, 0, 0, 1)
013 > p.coefficients
    => Vector[1, 0, 0, 1, 1, 0, 0, 0]
014 >
```

Задача 2

За решението на тази задача, дефинираме клас *RM* с полета *r*, *m* и *matrix*, съдържащи съответно редът, дължината и пораждащата матрица на кода. Инстанция на така зададения клас може да се създаде чрез извикване на метода му *RM#new(r, m)*.

Извеждане на всички кодове с дължина m

При зададено $m = 10$, изходът от програмата е във вида $RM(r_i, m)$, $[n, k, d]$, corrects t.

```
020 > require './rm.rb'
    => false
021 > m = 10
    => 10
022 > m.times do |r|
023?>   rm = RM.new r,m
024?>   p "%s, %s, corrects %s" %[rm, rm.notation, rm.corrects]
025?> end
"RM(0, 10), [1024, 1, 1024], corrects 511"
"RM(1, 10), [1024, 11, 512], corrects 255"
"RM(2, 10), [1024, 56, 256], corrects 127"
"RM(3, 10), [1024, 176, 128], corrects 63"
"RM(4, 10), [1024, 386, 64], corrects 31"
"RM(5, 10), [1024, 638, 32], corrects 15"
"RM(6, 10), [1024, 848, 16], corrects 7"
"RM(7, 10), [1024, 968, 8], corrects 3"
"RM(8, 10), [1024, 1013, 4], corrects 1"
"RM(9, 10), [1024, 1023, 2], corrects 0"
    => 10
026 >
```

Извеждане на параметрите на код по устойчивост на грешки

Методите *RM#correcting(t, min_d)* и *RM#detecting(t, min_d)*, където t е брой грешки, а min_d – минимална размерност на кода, изчисляват реда и дължината на най-оптималния код, който покрива изискванията и връщат инстанция на обекта *RM* създадена с изчислените параметри.

```
001 > require './rm.rb'
    => true
014 >   code = RM.detecting(5, 16)
    => RM(2, 5)
015 > code.notation
    => [32, 16, 8]
016 > code.corrects
    => 3
017 > code.detects
    => 7
018 >
```

Задача 3

За да решим задачата, създаваме клас *ReedCoder* с поле *code*, съдържащо кода на Рид-Малер, използван от кодера. Инстанция на този клас може да се създаде чрез извикване на метода *ReedCoder#new(code)*.

Кодиране на информационен вектор

Методът *ReedCoder#encode_vector(vector)* изчислява кодовия вектор c чрез умножение на информационния вектор v с пораждащата матрица G на използвания код на Рид-Малер $c = v \times G$.

```
001 > require './reed_coder.rb'
       => true
002 > require './rm.rb'
    => true
003 > code = RM.new 1, 4
    => RM(1, 4)
004 > coder = ReedCoder.new code
    => #<ReedCoder:0x00000002734ab8 @code=RM(1, 4)>
005 > coder.encode_vector [1, 0, 1, 1, 0]
    => Vector[1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1]
006 > coder.decode_vector [1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1]
    => Vector[1, 0, 1, 1, 0]
007 >
```

Декодиране на пристигнал вектор

Методите *ReedCoder#decode_vector(v)* и *ReedCoder#decode_matrix(m)* декодират съответно получен вектор и матрица от получени вектори, като *#decode_vector(v)* реализира несистематичния декодер на Рид, а *#decode_matrix(m)* се свежда до него.

```
001 > require './rm.rb'
    => true
002 > require './reed_coder.rb'
    => true
003 > v1 = [1,0,0,0,0,1,0,1,1,1,1,0,0,1,0,1]
004 > v2 = [1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1]
005 > v3 = [1,1,1,0] * 4
006 > code = RM.new 1, 4
    => RM(1, 4)
007 > coder = ReedCoder.new code
    => #<ReedCoder:0x00000002734ab8 @code=RM(1, 4)>
008 > coder.decode_vector v1
    => Vector[1, 0, 1, 1, 1]
009 > coder.decode_vector v2
    => Vector[1, 1, 1, 1, 1]
010 > coder.decode_vector v3
    => Vector[1, 0, 0, 0, 0]
011 >
```

Изходен код

polynom.rb

```
require 'matrix'

class Polynom
  attr_reader :coefficients, :power, :truth_table

  def Polynom.truth_vector(*vector)
    coefficients = Polynom.get_system(Math.log2(vector.size).to_i)
    constants = Matrix.column_vector vector
    solutions = (coefficients.inverse * constants).map {|e| e.to_int % 2}
    new solutions.column 0
  end

  def Polynom.[](*coefficients)
    new coefficients
  end

  def value_for(vector)
    @truth_table[vector]
  end

  def truth_vector
    Vector.elements @truth_table.values
  end

  private
  def is_pow2?(n)
    n & (n - 1) == 0
  end

  def is_binary?(n)
    n == 1 || n == 0
  end

  def initialize(coefficients, values=nil)
    raise "Please␣specify␣2^n␣coefficients" if !is_pow2? coefficients.size
    raise "Valid␣coefficients:␣0␣or␣1" if coefficients.any? {|n| !is_binary? n}

    @power = Math.log2(coefficients.size).to_i
    @coefficients = Vector.elements coefficients
    values = values ? values : Polynom.get_truth_vector(coefficients, @power)
    @truth_table = Hash[*Polynom.arg_table(@power).zip(values).flatten(1)]
  end

  def Polynom.get_combinations(variables, power=nil)
    power = power ? power : variables.size
    Array.new(power) do |key|
      variables.combination(key+1).to_a
    end.flatten(1)
  end

  def Polynom.multiply(variables, power=variables.size)
    combinations = Polynom.get_combinations(variables, power)
    [1] + combinations.map! do |combination|
      combination.reduce(:&)
    end
  end

  def Polynom.arg_table(vars)
```

```
    Array.new(2**vars) do |row|
      Array.new(vars) {|column| row[column]}
    end.sort {|a, b| a.count(1) <=> b.count(1)}
  end

  def Polynom.get_system(vars)
    Matrix.rows arg_table(vars).map {|args| multiply(args)}
  end

  def Polynom.get_truth_vector(coefficients, power)
    elements = arg_table(power).map do |values|
      mononoms = Vector.elements(Polynom.multiply values)
      values = mononoms.inner_product(coefficients) % 2
    end
    Vector.elements elements
  end
end
```

rm.rb

```
require 'matrix'
require './polynom.rb'

class RM
  attr_reader :r, :m, :matrix

  def initialize(r, m)
    raise "r must be >= 0 and m must be >= r" if r < 0 or r > m
    @r, @m = r, m
    calculate_gen_matrix
  end

  def detects
    weight - 1
  end

  def corrects
    (weight - 1)/2
  end

  def information_rate
    Rational(dimension, length)
  end

  def notation
    [length, dimension, weight]
  end

  def to_s
    "RM(%d, %d)" % [@r, @m]
  end

  #n
  def length
    2**@m
  end
  alias :size :length

  #k
  def dimension
    matrix.row_size
  end
```

```
  # Minimum Hamming weight
  # d_min
  def weight
    2 ** (@m-@r)
  end

  def ==(other)
    other.r == @r && other.m == @m
  end

  def RM.detecting(errors, min_dimension)
    difference = Math.log2(errors + 1).ceil
    RM.by_parameters(difference, min_dimension)
  end

  def RM.correcting(errors, min_dimension)
    difference = Math.log2(Rational(1, 2) + errors).ceil + 1
    RM.by_parameters(difference, min_dimension)
  end

  private

  def RM.by_parameters(difference, min_dimension)
    r, m = 0, difference
    begin
      rm = RM.new(r, m)
      if m - r > difference
        r+=1
      else
        r = 0
        m += 1
      end
    end while rm.dimension < min_dimension
    rm
  end

  def calculate_gen_matrix
    arguments = Array.new(2**@m) do |row|
      Array.new(@m) { |column| row[@m - column - 1] }
    end
    columns = arguments.map {|row| Polynom.multiply row, @r}
    @matrix = Matrix.columns columns
  end
end
```

reed_coder.rb

```
module Math
  def Math.factorial(n)
    1.upto(n).inject(1) {|result, element| result * element}
  end
  def Math.choose(n, k)
    return 0 if k > n
    Rational(factorial(n), (factorial(k) * factorial(n-k)))
  end
end

module Statistics
  def Statistics.mode(array)
    array.group_by {|value| value}.values.max_by(&:size).first
  end
end
```

```ruby
class ReedCoder
  attr_reader :code

  def initialize(code)
    @code = code
  end

  def encode_matrix(input)
    input.to_a.map {|vector| encode_vector vector}
  end

  def encode_vector(vector)
    result = Matrix.row_vector(vector) * code.matrix
    result.row(0).map {|element| element % 2}
  end

  def decode_vector(vector)
    result = []
    # Main loop -- decreasing the order of the code
    code.r.downto(0) do |order|
      # The number of symbols of the information vector that we will be able to
      # calculate from this order.
      symbols = Math.choose(code.m, order).to_i

      # The number of bits of the vector per checksum for this order
      monomials = 2**order

      # The number of checksums per symbol for this order
      checksums = vector.size / monomials

      symbols.downto(1) do |symbol|
        # Offset of the first bit of the vector that is used in the sum
        offset = 0

        # Distance between the bits of the vector used in the sums
        distance = 2**(symbols - symbol)

        # The size of a block of checksums
        block_size = monomials * distance

        # The number of blocks of checksums for this symbol
        blocks = vector.size / block_size

        sums = []
        blocks.times do |block|
          distance.times do
            sum = 0
            monomials.times do |monomial|
              sum += vector[offset + monomial*distance]
            end
            sums << sum
            offset += 1
          end
          offset = block * block_size
        end
        result << Statistics.mode(sums)
      end
      vector = adjust(vector, symbols, order)
    end
    Vector.elements result.reverse.map {|element| element % 2}
  end
```

```
  def adjust(vector, coefficients, power)
    number = Math.choose(@code.m, power).to_i
    offset = 0.upto(power).inject(0) do |result, k|
      result+= Math.choose(@code.m, k)
    end.to_i

    vector = Vector.elements vector
    number.times do |index|
      vector = vector + code.matrix.row(offset - 1 - index) * coefficients[index]
    end
    vector.map {|element| element % 2}
  end

  def decode_matrix(matrix)
    matrix.to_a.map {|vector| decode_vector vector}
  end
end
```

Спецификация на тестовете

```
Polynom
  gives you a vector of its coefficients
  only works with binary coefficients
  thorws an error when a vector of an incorrect size is given
  only accepts 2^n coefficients
  returns a correct value when given variables values to substitute with
  can tell its power
  constructs a correct argument table
  calculates a truth vector correctly
  construction
    can be done by passing a vector of coefficients
    can be done by passing a truth vector

ReedCoder
  construction
    can be done by passing an RM code
  encoding
    can be done to a vector
    can be done to a matrix
  decoding
    can be done to a vector
    can be done to a matrix
    should adjust the input vector accordingly

RM
  returns its length
  returns its dimension
  returns its minimum distance
  returns its standard code notation
  returns its information rate
  returns how many errors it can correct
  returns how many errors it can detect
  calculates the parameters of codes that can correct the passed number of errors
  calculates the perameters of codes that can detect the passed number of errors
  construction
    raises an error when a code does not exist for the given arguments
    cunstructs the generating matrix correctly

Finished in 0.0279 seconds
27 examples, 0 failures
```