Теория информации, решения задач части 2 (14-16) (линейные коды)

Table of Contents

1 2.14 Радиус покрытия

Итак, задача состоит в том чтобы подсчитать радиус покрытия кодов из задания 2.1 и определить, можно ли их улучшить (опираясь на леммы о радиусе покрытия). Код, подсчитывающий радиус, приведен в приложении 1. Для каждого вектора v_1 длины n он декодирует его в v_2, затем считает между ними вес и выводит максимальный. Проанализируем результаты, полученные на матрицах $H_1 \dots H_5$ (H_6 не существует, но улучшить код $(6, 6)$, очевидно, нельзя, поэтому опустим его анализ).

Итак, матрицы:

$$H_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} H_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$H_3 = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} H_4 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} H_5 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Сводная таблица характеристик (для кода C вектора $v_1 \notin C$ и $v_2 \in C$ – те, на которых достигается максимальная дистанция):

ix	n	k	r	d	ρ	v_1	v_2
H_1	6	1	5	6	3	111000	111111
H_2	6	2	4	4	3	111000	110101
H_3	6	3	3	3	2	111111	110011
H_4	6	4	2	2	2	111111	011110
H_5	6	5	1	1	1	111110	111111

Свойства радиуса покрытия таковы, что если $\rho > d$, то k можно увеличить без уменьшения d. В то же время $\rho \le r$. Отсюда следует, что для первого кода ($r < d$) улучшение невозможно в принципе. Для кодов 4 и 5 верно $\rho = d$, улучшение также невозможно. Оптимальность H_2 и H_3 очевидна и была доказана ранее в 2.1.

2 2.15 Декодирование по соседям нулевого слова

Код реализованного алгоритма находится в приложении 2. Он использует перебор по всем комбинациям векторов кодовых слов, начиная с малых. Альтернативная реализация (закомментирована в приложении) использует предпосылку о том, что множество Z можно найти жадно, убирая по элементу из множества всех векторов длины n. Насколько показали даннные, оба метода выдают один и тот же результат, но я не берусь доказывать корректность второго подхода.

Итак, в iй строке расположен Z для H_i.

```
[111111]
[011011,101110,110101]
[000111,011010,011101,101001,101110,110100]
[000011,000101,001001,010001]
[000011,000101,001001,010001,100001]
```

В приложении также приведены данные о том, какие слова входят в код, о решающей и ближайшей окрестности нуля (для того, чтобы убедиться в верности полученных данных).

Алгоритм декодирования по соседям нулевого слова является типичным примером оптимизации предподсчета – мы снижаем асимптотику запроса засчет некоторых готовых данных (в нашем случае Z).

Наивное декодирование по кодовым словам работает за $O(2^n)$. Декодирование перебором по комбинациям векторов ошибок (и его оптимизация синдромное декодирование) имеет сложность $O(2^r)$ (предподсчета, реальное время ответа на запрос – $O(1)$, если считать операици умножения на матрицу базовыми).

Декодирование по соседям нулевого слова занимает достаточно много времени на предподсчет – наивное вычисление Z предполагает $O(2^n)$ итераций проверки "можно ли не включать этот вектор в Z". Также, сама проверка достаточно дорогостоящая – она предполагает вычисление решающей области очередного кодового слова (конечно, это тоже можно предподсчитать). Асимптотика ответа на запрос хорошо и наглядно описана в параграфе 2.6.2, и (практически цитируя), начиная со скорости $R = 0.1887$ декодирование по соседям нулевого слова становится гораздо более эффективным чем предыдущие два метода. Для всех кодов кроме $(6, 1)$ скорость R выше этой границы.

3 2.16 Декодирование по информационным совокупностям

Будем проверять все комбинации размера 4 строк G на линейную независимость. Код в приложении 3. Для начала найдем G кода Хэмминга (7,4):

$$G_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Алгоритм выдал для нее 16 наборов ЛНЗ столбцов, то есть количество информационных совокупностей.

Матрица H расширенного кода Хэмминга (8,4) приведенная к систематическому виду оказалась равной матрице G:

$$G_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

И она также имеет 16 информационных совокупностей.

Опять-таки, ссылаясь на доказательства в главе 2.6.3, декодирование по информационным совокупностям будет быстрее для любого R, так как асимптотическая оценка показателя сложности декодирования строго меньше, чем у всех рассмотренных ранее способов декодирования.

4 Приложение 1

```
-- | Given matrix H, returns (r,v1,v2) -- code radius r, vector v1
-- (not in code) and vector v2 (in code) such that w(v1-v2) = r.
codeRadius :: [BVector] -> (Integer, BVector, BVector)
codeRadius h =
    maximumBy (comparing $ view _1) $
    map (\y -> let d = decode y in (weight (decode y `sumBVectors` y), y, d))
        (binaryVectors $ fromIntegral n)
  where
    decode :: BVector -> BVector
    decode y = do
        let syndrom = y `vMulM` transpose h
        let e = decodeMap ! syndrom
        y `sumBVectors` e
    decodeMap = syndromDecodeBuild h
    n = length h
```

5 Приложение 2

Код реализованного алгоритма нахождения множества соседей нулевого слова (сам алгоритм декодирования пишется тривиально в 3 строки):

```
buildZeroNN :: [BVector] -> [BVector]
buildZeroNN h =
    traceShow (map showVec d0) $
    traceShow (map showVec $ solvingArea zero) $
    fromMaybe (error "should exist") $ find zCondition $ allCombinations $ codeH h
  where
-- -- Eager calculation.
--    kickWhilePossible $ delete zero $ codeH h
--  where
--    kickWhilePossible :: [BVector] -> [BVector]
--    kickWhilePossible zCandidate =
--        case find (\e -> zCondition $ delete e zCandidate) zCandidate of
--            Just x  -> kickWhilePossible (delete x zCandidate)
--            Nothing -> zCandidate
    zCondition :: [BVector] -> Bool
    zCondition zCandidate =
        let union = HS.fromList $ concat $ map solvingArea zCandidate
        in all (`HS.member` union) d0

    n = length h

    zero = replicate n False
```

```haskell
    d0 = neighborhood $ solvingArea zero

    solvingArea :: BVector -> [BVector]
    solvingArea a = filter ((== a) . decode) $ binaryVectors n

    -- decoding
    syndromMap = syndromDecodeBuild h
    decode :: BVector -> BVector
    decode y = do
        let syndrom = y `vMulM` transpose h
        let e = syndromMap ! syndrom
        y `sumBVectors` e

    -- Calculates closest neighborhood by solving area
    neighborhood :: [BVector] -> [BVector]
    neighborhood sA = filter (\x -> x `notElem` sA && invertedIn x)
                             (binaryVectors (length $ unsafeHead sA))
      where
        invertedIn x =
            let invertedSet =
                    mapMaybe (\i -> if x !! i then Just (x & ix i .~ False) else Nothing)
                             [0..length x-1]
            in any (`elem` sA) invertedSet
```

Промежуточные данные для $H_1 \dots H_5$: C, решающая область нуля и ближайшая окрестность нуля:

```
H1:
[000000,111111]
[000000,000001,000010,000011,000100,000101,000110,000111,001000,001001,001010,
 001011,001100,001101,001110,010000,010001,010010,010011,010100,010101,010110,
 011000,011001,011010,011100,100000,100001,100010,100100,101000,110000]
[001111,010111,011011,011101,011110,100011,100101,100110,100111,101001,101010,
 101011,101100,101101,101110,110001,110010,110011,110100,110101,110110,111000,
 111001,111010,111100]

H2:
[000000,011011,101110,110101]
[000000,000001,000010,000011,000100,000101,000110,000111,001000,001001,001010,
 001100,001101,010000,010100,100000]
[001011,001110,001111,010001,010010,010011,010101,010110,010111,011000,011001,
 011010,011100,011101,100001,100010,100011,100100,100101,100110,100111,101000,
 101001,101010,101100,101101,110000,110100]

H3:
[000000,000111,011010,011101,101001,101110,110011,110100]
[000000,000001,000010,000100,001000,001100,010000,100000]
[000011,000101,000110,001001,001010,001101,001110,010001,010010,010100,011000,
 011100,100001,100010,100100,101000,101100,110000]

H4:
[000000,000011,000101,000110,001001,001010,001100,001111,010001,010010,010100,
 010111,011000,011011,011101,011110]
[000000,000001,100000,100001]
[000010,000011,000100,000101,001000,001001,010000,010001,100010,100011,100100,
 100101,101000,101001,110000,110001]

H5:
[000000,000011,000101,000110,001001,001010,001100,001111,010001,010010,010100,
 010111,011000,011011,011101,011110,100001,100010,100100,100111,101000,101011,
 101101,101110,110000,110011,110101,110110,111001,111010,111100,111111]
[000000,000001]
[000010,000011,000100,000101,001000,001001,010000,010001,100000,100001]
```

6 Приложение 3

Код, вычисляющий количество информационных совокупностей.

```
linearDependent :: [BVector] -> Bool
linearDependent [] = False
linearDependent vectors
    | any (== zero) vectors = False
    | otherwise = or $ map ((== zero) . sumAll) ps
  where
    n = length $ unsafeHead vectors
    zero = replicate n False
    sumAll :: [BVector] -> BVector
    sumAll = foldr sumBVectors (replicate (length $ unsafeHead vectors) False)
    ps :: [[BVector]]
    ps = allCombinations vectors

task216 :: IO ()
task216 = do
    let rankk k x = length $ filter (not . linearDependent) $ combinations k x
    print  $ rankk 4 hamming74G
    print  $ rankk 4 hamming84G
```

Author: Волхов Михаил, M4139
Created: 2017-11-21 Tue 07:52