ハフマン符号を解く

田中 哲朗（東京大学情報基盤センター）
ktanaka@tanaka.ecc.u-tokyo.ac.jp

■問題の定義

今回取り上げる問題は2002年度のGreater New York地区予選の問題G「Dehuff」である．

これまで，連載では主にアジア予選や決勝大会の問題を取り上げてきたが，今回は他地区の地区予選[☆1]も含めて，問題を探してみることにした．

問題と最終結果だけが公開されている予選大会が多い中で，Greater New York地区予選では，判定に使われた入力や，想定される出力，各チームの問題ごとの解答状況などさまざまなデータが公表されている．面白い問題が出されることも多いので，ウォッチャーとしては見逃せない大会となっている．

それでは問題をみてみることにしよう．今回の問題は可変符号化によるデータ圧縮を扱っているので，問題の背景となる知識も補って説明する．

アルファベット（文字集合）中の文字を可変長（1ビット以上）のビット列（符号語）に対応させて，符号化することを考える．1文字分のビット列を受け取った時点で復号化を可能とするためには，任意の文字に対応する符号語が，別の文字に対応する符号語の頭から何ビットか切り出した語頭（prefix）とは一致しないという語頭条件（prefix condition）を満たす必要がある．語頭条件を満たす符号のことを語頭符号と呼ぶ．

出現頻度に偏りがある文字集合を符号化する際には，出現頻度が高い文字は短いビット長で，低い文字は長いビット長で符号化すると，平均的に少ないビット長で符号化が行える（各文字をすべて同じビット長で表現していた場合は，この符号化の結果，短いビット長で表されるので圧縮になっている）．

出現頻度が与えられたときに，平均ビット長を最短にする語頭符号の構成法は1952年にDavid Huffmanによって提案された．

出現頻度が低い順に文字を2つ取り去り，この2つを融合した仮想的な文字（出現頻度は取り去った文字の出現頻度の和）を加えるステップを繰り返すと最後に文字が1つだけ残る．

この文字の符号語を空ビット列「」とする．2つの文字を融合した仮想的な文字の符号語に対して，融合前の文字の片方をこれにビット「0」を連結したビット列，もう片方をビット「1」を連結したビット列として定義すると，すべての文字に対応するビット列を定義できる．構成法からいって，得られる符号が語頭符号になっていることはあきらかである．また，この構成法で構成した1文字に対応するビット列の長さの最大値は「文字種数（アルファベットのサイズ）−1」以下であることもいえる．

今回の問題「Dehuff」は，符号化前の文字列と，語頭符号によって符号化されたビット列が与えられたときに，符号表を推定するという問題である．

たとえば，「ABC」を符号化して，「11100」になったとする．1文字は必ず長さ1以上のビット列に対応するので，[1,1,100], [1,11,00], [1,110,0], [11,1,00], [11,10,0], [111,0,0]の6種類の可能性があるが，語頭条件を満たすのは，

```
A = 11
B = 10
C = 0
```

のみなので，これが答えになる．

問題には，

- 問題で想定されるアルファベットは英大文字とスペースの部分集合．
- 符号化前の文字列ではアルファベット中のすべての文字が1回以上使われている．

☆1 2003年大会では世界中の31カ所で地区予選が開催される．

• 符号空間全体を使っている（アルファベット中の文字に対応しないビット列はない）.

という条件が加わっている．また，入力は最初に問題数が入り，後は符号化前の文字列，符号化後のビット列が 1 行ずつ続く.

問題には符号の頻度をもとに符号表を作成するという仮定がないので，語頭符号であってもハフマン符号とはいえない．しかし，適当な頻度表を仮定すれば，それにしたがって作成されたハフマン符号と見なすことが可能なので，元の問題名「Dehuff」を尊重して，今回の記事は「ハフマン符号を解く」というタイトルにした.

プログラムコンテストに C 言語で（あるいは C++ 言語でも template library を使わずに）挑むのは不利だというのが筆者の持論なので，C++ 言語で STL を使いまくってプログラムを書くことにする.

■入力処理と main

C++ 言語は C 言語や Java 言語と比較すると少ないタイピング量で入力ルーチンを書くことができる．その点はプログラムコンテスト向きだが，今回のように，空白文字も文字として扱わなければいけない場合は注意が必要となる．is を入力ストリームとして，

```
string original;
is >> original;
```

のようにして文字列 original に読み込むと，

```
HELLO WORLD
```

のような入力行に対しては，「HELLO」だけしか読み込まれない．行単位の入力を行うには，

```
getline(is,original);
```

のようにする必要がある.

以下のように，main 関数は入力を行うだけで解法部分は関数 dehuff にまかせることにする.

```
int main(int ac,char **ag){
  // パラメータなしで呼び出したときのファイル名
  char *filename="problemg.in";
  // パラメータつきの場合
  if(ac>1) filename=ag[1];
  // 入力ストリームを開く
  ifstream is(filename);
  // 問題数を読む
  int n;
  is >> n;
  // 問題数を読んだ後の改行を読み飛ばす
  string dummy;
  getline(is,dummy);
  for(int i=1;i<=n;i++){
    // 符号化前の文字列を読む
    string orig;
    getline(is,orig);
    // 符号化後のビット列（の文字列表現）を読む
    string coded;
    getline(is,coded);
    // 解読を行い結果を表示する.
    dehuff(i,orig,coded);
  }
  return 0;
}
```

上のプログラムで，

```
string dummy;
getline(is,dummy);
```

の部分が気になる人がいるかもしれない．これは，

```
is >> n;
```

を実行したあと，is が次に読み込む文字が，ストリーム中の次の非数字文字である改行になっているので，これをストリームから取り除くためのものである．これを入れておかないと，最初の行入力として空行を読み込んでしまう.

■符号表の表現

語頭符号の符号表は，図 -1 のように二進木で表現するのが自然である．図 -1 は

```
A = 00
B = 010
C = 011
D = 1
```

という符号表に対応している.

しかし，ここではなるべく STL を使って書くという方針なので，連想配列 map を使って，

```
typedef map<char,string> CodeTable;
```

で定義される型 CodeTable で符号表を表現することにする．こうすると

```
CodeTable codeTable;
codeTable['A']="00";
codeTable['B']="010";
```

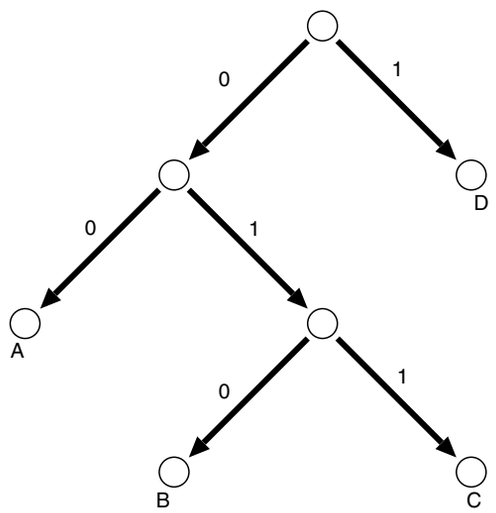

図-1　二進木による符号表の表現

```
codeTable['C']="011";
codeTable['D']="1";
```

のように簡単に符号表を作り出すことができる☆2.

■符号空間全体を使っているか

語頭符号だというだけでは問題の条件を満たすとはいえない．たとえば，

```
ABC
011011
```

から，

```
A 01
B 10
C 11
```

という符号表ができたとする．これは，語頭符号だが，「00」というビット列に対応するアルファベット中の文字はないので，符号空間全体を使っているという条件を満たしていない．

符号表中の i 番目のビット列の長さを l_i としたとき，符号空間全体を使っている場合は，

$$\sum \frac{1}{2^{l_i}} = 1$$

という式を満たしているはずである．これをチェックするプログラムは以下のようになる．

☆2 語頭条件のチェックに関しては二進木よりも計算量がかかるが，全体から考えると無視できる．

```
// 符号表全体を使っているかどうかのチェック
bool checkFullSpace(CodeTable const& ct){
  // ビット列のビット数を記録する優先度付きキュー
  priority_queue<int> pq;
  // ビット数を順に記録
  for(CodeTable::const_iterator
        it=ct.begin();
      it!=ct.end();++it){
    pq.push(it->second.size());
  }
  assert(!pq.empty());
  while(true){
    // 最大のビット数を返す
    int one=pq.top();
    pq.pop();
    if(pq.empty()){
      // 1つになったときは，0であるはず
      if(one!=0) return false;
      return true;
    }
    int two=pq.top();
    pq.pop();
    // 最大の2つは同じビット数のはず
    if(one!=two) return false;
    // 2つを融合した文字のビット数はone-1
    pq.push(one-1);
  }
}
```

なお，この問題では，アルファベットは大文字とスペースのみからなるので，集合の要素数は 27 以下である．そこで，32 ビットの符号なし固定小数点数（整数部分を 1 ビット，小数点以下を 31 ビット）で表現して計算すると，以下のように簡単に書ける．

```
bool checkFullSpace(CodeTable const& ct){
  unsigned int sum=0;
  for(CodeTable::const_iterator it=
        ct.begin();
      it!=ct.end();
      ++it){
    sum+=0x80000000>>(it->second.size());
  }
  if(sum==0x80000000) return true;
  return false;
}
```

整数部分が 1 ビットしかないので，桁あふれが気になる人もいるかもしれないが，この場合は問題ない．語頭符号であることが保証されている場合，$\sum \frac{1}{2^{l_i}} \leq 1$ なので，加算により桁があふれることはないからだ．

■単純なバックトラックによる解法

プログラミングコンテストでは，「単純なアルゴリズムでクリアできる問題では凝った工夫をしない」というのが鉄則となっている．まずは以下のような単純なバックトラックによる解法を試してみる．

1. 符号化前の文字列を前から1文字ずつ見ていく．
2. 出現が2度目以降の文字の場合は，符号化後のビット列の先頭が符号表中のビット列と一致するかどうかをチェックする．一致しない場合は，バックトラックする．一致する場合は次の文字に進む．
3. 初めて出現する文字の場合は，ビット列のビットを長さ1から長さ「文字種数 −1」まで切り出してその文字に対応するビット列の候補とする．語頭条件を満たさなくなる場合を除いた候補すべてについて，その候補を符号表に入れた上で，次の文字を試していく．

この解法を再帰関数を使って書くと以下のように書ける．

```
/**
 * orig : 符号化前の文字列
 * coded : 符号化後のビット列
 * alphabetSize : 文字種数
 * codeTable : すでに確定済みの符号表
 * ret : 解の符号表をvectorにしてためていく
 */
void makeCodeTable(string const& orig,
                   string const& coded,
                   size_t alphabetSize,
                   CodeTable & codeTable,
                   vector<CodeTable>
                   &ret){
  // 残りが0のとき
  if(orig.size()==0 && coded.size()==0){
    // 符号空間全体を使っているか？
    if(checkFullSpace(codeTable)){
      ret.push_back(codeTable);
    }
    return;
  }
  // どちらかが余った場合
  if(orig.size() > coded.size() ||
     orig.size()==0)
    return;
  // 最初の1文字
  char firstChar=orig[0];
  CodeTable::const_iterator it=
    codeTable.find(firstChar);
  // 出現済みの場合
  if(it!=codeTable.end()){
    // 符号化した結果
    string const& s=(*it).second;
    size_t len=s.size();
    // codedの先頭と一致しない場合は可能性がない
    if(coded.compare(0,len,s)!=0) return;
    // 残りの部分を再帰呼び出し．
    makeCodeTable(string(orig,1),
                  string(coded,len),
                  alphabetSize,codeTable,
                  ret);
  }
  // 初めて出現する文字
  else{
    // 符号化した結果のビット列のビット数
    for(size_t i=1;
        i<=min(alphabetSize-1,
        coded.size());
        i++){
      // 語頭条件のチェック
      // 符号表中のすべてについて，
      for(CodeTable::const_iterator it=
            codeTable.begin();
          it!=codeTable.end();++it){
        string const& s=(*it).second;
        size_t len=min(s.size(),i);
        // sの0ビット目からlenビットと
        // codedの0ビット目からlenビットを比較
        if(s.compare(0,len,coded,0,len)==
           0)
          goto noMatch;
      }
      // codedからiビットを符号として試してみる
      tryCode(orig,coded,i,alphabetSize,
              codeTable,ret);
      // 解が複数になった場合はそれ以上調べない
      if(ret.size()>1) return;
    noMatch:;
    }
  }
  return;
}
void tryCode(string const& orig,
             string const& coded,
             int codedLen,
             size_t alphabetSize,
             CodeTable & codeTable,
             vector<CodeTable> &ret){
  // 符号表を追加
  char firstChar=orig[0];
  codeTable[firstChar]=
    string(coded,0,codedLen);
  string newOrig(orig,1);
  string newCoded(coded,codedLen);
  // 残りの部分を再帰呼び出し．
  makeCodeTable(newOrig,newCoded,
                alphabetSize,codeTable,
```

```
                ret);
    // 符号表を元に戻す
    codeTable.erase(firstChar);
  }
```

上のプログラムの，string(coded,0,codedLen) のような部分は STL に不馴れな人は分かりにくいかもしれない．この string のコンストラクタは，

string(const string& str, size_type pos, size_type n)

文字列 str の pos 文字目から n 文字切り出した部分文字列を作る．第 3 引数が省略された場合は，文字列 str の最後まで．

ということを意味している．上のように文字列のコピーや部分文字列を何度も繰り返すコードは，C 言語プログラマには重そうで耐えがたいかもしれないが，数十，数百文字程度の文字列を扱う場合は，気にするほどではない．また，どうしても気になる場合は，変更をしない限り文字列本体はコピーしない string の実装を探し出してきて使えば，上のプログラムでは実体のコピーはほとんど発生しない[☆3]．

なお，文字列をスキャンして，出現する文字種の数を求める関数は以下のように書ける．

```
int charSetSize(string const& orig){
  // 文字の集合chars
  set<char> chars;
  // orig中の文字を集合charsにコピー
  copy(orig.begin(),orig.end(),
       inserter(chars,chars.begin()));
  // 集合の大きさが文字種の数
  return chars.size();
}
```

main から呼ばれる 関数 dehuff は以下のように，関数 makeCodeTable に対して文字列，符号化後のビット列と，文字種数を渡す．ret のサイズが 1 なら唯一の解，2 以上なら複数解，0 なら解なしとなる．問題文では解は必ずあるとして，解がない場合の出力形式を規定していないが，デバッグのため「NO ANSWER」と表示することにする．

```
void dehuff(int n,string const& orig,
            string const& coded){
  cout << "DATASET #" << n << endl;
  size_t alphabetSize=charSetSize(orig);
  vector<CodeTable> ret;
  makeCodeTable(orig,coded,alphabetSize,
                codeTable,ret);
  int ansCount=ret.size();
  if(ansCount==0){
    cout << "NO ANSWER" << endl;
  }
  else if(ansCount>1){
    cout << "MULTIPLE TABLES" << endl;
  }
  else{
    CodeTable &result=ret[0];
    for(CodeTable::const_iterator
          it=result.begin();
        it!=result.end();
        ++it){
      cout<< (*it).first << " = " <<
             (*it).second << endl;
    }
  }
}
```

■逆方向のスキャンによる枝刈り

元の文字列の最後の方に符号表中の文字があるときは，符号化後のビット列の最後が符号表と一致するかチェックすれば，間違った符号表を早めに検出できる．たとえば，符号表が

```
A = 10
B = 010
```

となっていて，元の文字列が

```
....BA
```

で終わっているとする．符号化後のビット列が

```
....01010
```

で終わっていればよいが，

```
....01110
```

で終わっているとすると，解には成り得ないことが分かる．なお一致した場合は，一致した部分を符号化前の文字列と，符号化後のビット列の両方から取り除くことができる．

この改良は tryCode を少し変更することで実現できる．

```
void tryCode(string const& orig,
             string const& coded,
             int codedLen,
             size_t alphabetSize,
```

☆3 残念ながら手元の処理系では常にコピーを行っているようだ．

```
            CodeTable & codeTable,
            vector<CodeTable> &ret){
  // 符号表を追加
  char firstChar=orig[0];
  codeTable[firstchar]=
    string(coded,0,codedLen);
  string newOrig(orig,1);
  string newCoded(coded,codedLen);
  while(newOrig.size()>0){
    // 最後の文字が符号表から見つかるか
    CodeTable::const_iterator it=
      codeTable.find(
        newOrig[newOrig.size()-1]);
    // 見つからなかったらやめる
    if(it==codeTable.end())break;
    string const& coded1=it->second;
    size_t codeLen=coded1.size();
    // 残りが短すぎるとうまくいかない
    if(newCoded.size()<codeLen)
      goto fail;
    size_t restSize=
      newCoded.size()-codeLen;
    // 終りが符号表と一致しないとうまくいかない
    if(newCoded.compare(restSize,
                        coded1.size(),
                        coded1)!=0)
      goto fail;
    newCoded=string(newCoded,0,restSize);
    newOrig=string(newOrig,0,
                   newOrig.size()-1);
  }
  // 残りの部分を再帰呼び出し.
  makeCodeTable(newOrig,newCoded,
                alphabetSize,codeTable,
                ret);

  // 符号表を元に戻す
fail:
  codeTable.erase(firstChar);
}
```

■ビット列の長さによる枝刈り

残りの文字列に対応する符号化後のビット列の長さは，符号表中にある文字に関してはその長さ，ない文字に関しては長さ 1 として総和をとったビット数以上となる．この制限値は以下のようにして簡単に計算できる．

```
size_t minCodedSize(string const& orig,
                    CodeTable const&
                    codeTable){
  size_t len=0;
  for(size_t i=0;i<orig.size();i++){
    CodeTable::const_iterator it=
      codeTable.find(orig[i]);
    if(it==codeTable.end()){
      len+=1;
    }
    else len+=(it->second).size();
  }
  return len;
}
```

makeCodeTable の最初で，

```
  if(minCodedSize(orig,codeTable) >
     coded.size())
    return;
```

のようにチェックをすることで，可能性のない場合に探索を打ち切ることができる．

符号表中にない文字に関して，すべて 1 ビットと見積もっているのは，ちょっと大ざっぱである．もっと精密に見積もることも可能だが，これだけでもかなり効果があることが期待される．

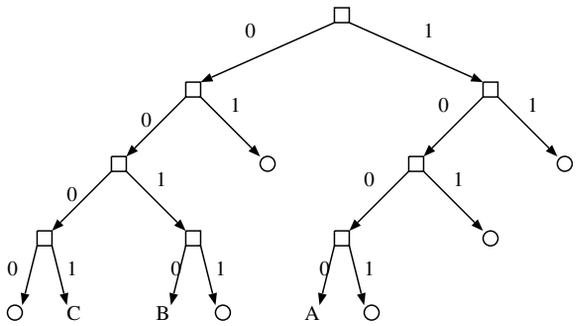

図 -2　構成途中の符号表

■符号空間全体を使っているかによる枝刈り

ここまでのプログラムでは，符号空間全体を使っているかどうかのチェックを，すべての文字に対応する符号表を作ってから行っていた．しかし，途中で符号空間全体を使っている符号表を作れないことが判明すれば，探索を打ち切ることができる．

たとえば，アルファベット $\{A,B,C,D,E,F,G,H\}$ について，A, B, C の符号表を

```
A = 1000
B = 0010
C = 0001
```

と作ったとする．図 -2 のように，割り当ての決まっ

ていない部分（ここでは便宜的に穴と呼ぶ）を丸で表した二進木で表せる．この図には穴が6個あるが，文字に対する符号を決定した場合に，穴は高々1個ずつしか埋められないので，残り5個の文字の割り当てを行った後にも穴が残ってしまうことになる．

符号表を二進木で表現する場合は自然に穴を表現できるが，mapで表現する場合は，ちょっと工夫がいる．

まず，符号表中のすべてのビット列の語頭を取り出す．

```
  A = 1000 -> {1, 10, 100, 1001}
  B = 0010 -> {0, 00, 001, 0010}
  C = 0001 -> {0, 00, 000, 0001}
-> {0,1,00,10,000,001,100,0001,0010,1001}
```

それぞれの，最後の1ビットを反転したものを得る．

```
   {0,1,00,10,000,001,100,0001,0010,1001}
-> {1,0,01,11,001,000,101,0000,0011,1000}
```

このうちで，元の語頭に含まれないものが，穴になる．

```
{1,0,01,11,001,000,101,0000,0011,1000}-
   {0,1,00,10,000,001,100,0001,0010,1001}
-> {01,11,101,0000,0011,1000}
```

この部分のチェックをmakeCodeTableの関数の先頭に，

```
if(!holeCheck(codeTable,alphabetSize))
  return;
```

のように入れることにする．

関数holeCheckは以下のように書ける．

```
char binaryNot(char c){
  if(c=='1') return '0';
  else return '1';
}

/**
 * codeTable中の穴の数を数えて
 * alphabetSize以下であるかどうかをチェックする．
 */
bool holeCheck(CodeTable const&
                 codeTable,
               size_t alphabetSize){
  // prefix部分の集合
  set<string> prefixes;
  // prefix部分の最下位を反転した集合
  set<string> notPrefixes;
  for(CodeTable::const_iterator it=
        codeTable.begin();
      it!=codeTable.end();++it){
    string const& s=it->second;
    for(size_t i=1;i<=s.length();i++){
      prefixes.insert(string(s,0,i));
      notPrefixes.insert(string(s,0,i-1)+
                         binaryNot
                         (s[i-1]));
    }
  }
  // 集合の差を求める -> 穴の数
  set<string> diff;
  set_difference(notPrefixes.begin(),
  notPrefixes.end(),
                 prefixes.begin(),
                 prefixes.end(),
                 inserter(diff,diff.
                 begin()));
  // 穴の数が残りの符号数よりも大きいとダメ
  if(diff.size()>alphabetSize- codeTable.
  size())
    return false;
  return true;
}
```

ここまで実現すると，無駄な探索はかなり減らすことができる．

■性能評価と問題点

はじめで触れたように，今回の問題では判定に使われた入力データが公開されているので，それをプログラムの入力に与えて，それぞれの改良を評価してみることにした．

単純なバックトラックのみによるプログラム（1）をベースにする．ベースプログラムに対して，以下の3つの改良はそれぞれ独立に適用可能である．

(a) 逆方向のスキャンによる枝刈り
(b) ビット列の長さによる枝刈り
(c) 符号空間全体を使っているかによる枝刈り

この2^3=8通りの組合せについて，判定に使われた入力データを与えて，makeCodeTableの呼び出し回数，実行時間（Athlon XP 3000+, 1GB, gcc 3.2.2, −O3）を計測した．結果を**表-1**に示す．

結果を見ると，(a), (b), (c) はそれぞれ効果があり，特に (c) は単独でも，けた違いに探索時間を減らす効果があることが分かった．組み合わせても効果があるが，(c) を適用済みのときに (b) を加えてそれほど効果がない．

しかし，プログラムの出力を公開されている出力データと比較してみたところ，出力結果が異なる問題が

プログラム	呼び出し回数	実行時間(s)
(1)	3944859	6.91
(1)+(a)	238366	0.47
(1)+(b)	282553	0.63
(1)+(c)	7526	0.18
(1)+(a)+(b)	201872	0.43
(1)+(a)+(c)	3169	0.07
(1)+(b)+(c)	7514	0.18
(1)+(a)+(b)+(c)	3169	0.06

表 -1　**makeCodeTable** の呼び出し回数，実行時間

プログラム	呼び出し回数	実行時間(s)
(1)	NA	>21600
(1)+(a)	119302155	345.73
(1)+(b)	10721559	31.61
(1)+(a)+(b)	951307	2.57

表 -2　**makeCodeTable** の呼び出し回数，実行時間
（符号空間全体を使っているかのチェックなし）

4 問見つかった．いずれも，公開された出力データでは，「MULTIPLE TABLES」となっているが，作成したプログラムでは，「NO ANSWER」と出力している．

調べてみると，問題文中の例題にも同種の問題が含まれていることが分かった．

```
ABCDEFGHI
01011011101111011111011111101111111000101
1111111
```

という入力がある．この入力に `makeCodeTable` を適用すると，`ret.size()` は 0 となり，解がないという結果が得られる．9 文字が 1 回ずつ現れる文字列を符号空間全体を使って符号化した場合のビット列の最大ビット数は，1+2+3+4+5+6+7+8+8 = 44 となるが，問題文での出力は 48 ビットなので当然の結果である．

しかし，問題文では例題に対しては「MULTIPLE TABLES」，複数の解が存在するという出力が提示されている．符号空間全部を使っているという条件を外せば，

```
ABBC-CD--DE---EF----FG-----GH----------HI------I
01011011101111011111011111101111111000101111111
A-AB-BC--CD---DE----EF-----FG------GH--HI------I
```

のように複数の解が存在するのはたしかだが，これまでの前提とは一致していない．

これをどう解釈すればよいだろうか．2 つの可能性を考えてみることにする．

1 つ目の解釈は，問題文は正しいが，例題が間違っているという考え方がある．

> you should be able to generate at least one binary code table for each letter in the alphabet.

とあるので，符号空間全体を使っている解が存在しない入力を作ってしまったのはミスであるというものだ．ただし，この場合は解がない場合に，「MULTIPLE TABLES」と出力するようにプログラムを作らないと，審判のチェックは通らない．

2 つ目の解釈は，問題文中に

> Note: For a given alphabet, the *entire* code space will be used; that is, there will be no unused codes.

とあるのが間違っているというものである．この観点に立って，プログラムを見直すと，

```
for(size_t i=1;
    i<=min(alphabetSize-1,coded.size());
    i++){
```

の部分で用いている符号化後のビット列の長さが「文字種数 −1」以下になるという仮定を用いることができなくなる．また，符号空間全体を使ったかどうかのチェックを行わないことになる．これは，枝刈りの効率をかなり下げることになる．この方針で書き直したプログラムを走らせてみた結果が**表 -2** である（実験に使った計算機環境は表 -1 と同じ）．バックトラックのみでは，6 時間経過しても答えが得られなかった．

どちらの解釈にしたがっても釈然としない結果になる．この問題については 3 チームが延べ 8 回解答を提出したという記録が残っているが，これらのチームもこの点に引っかかってしまって解答できなかったのではないだろうか．問題としては面白いものになり得ただけに残念なことである．

（平成 15 年 6 月 15 日受付）