Perfect-One-Error-Correcting Codes on a Family of N Dimensional Graphs

Paul Cull
Department of Computer Science
Oregon State University
Corvallis, Oregon
pc@cs.orst.edu

Jessica E. Cavanaugh
Siena College
Loudonville, NY
sjc4639@siena.edu

Kevin Stoller
Oregon State University
Corvallis, Oregon
stollerk@ucs.orst.edu

REU Program
Oregon State University

August 1997

Chapter 1

1.1 Introduction

The field of error-correcting codes is one in which there has been considerable research activity in the past three decades. One reason being that these codes are useful in a variety of areas from algorithm design to communication. In this paper, we present codes based on one, two, and three dimensional graphs, rather than the traditional hypercube representation. These codes can be easily constructed and decoded, which allows for an easier analysis. Also, the codes in two and three dimensions are intuitive and can be easily described by recursively constructed graphs. We examine a labeling for the one-dimensional graph, which is an alternative to the standard binary ordering, that results in simple error-correcting and decoding procedures. We show a codeword structure and conjecture for a labeling on the three dimensional graph, which is directly related to the alternate one-dimensional labeling. A proof that this structure forms a perfect-one error-correcting code for all $n \geq 0$. Another major result is a proof of the uniqueness of codeword structure on the graph for dimension n, since it is only been proven, to this point, on the one and two-dimensional graphs. We will also present some conjectures and questions which arose from our research.

1.2 Background

Throughout the paper, we will be using some standard symbols repeatedly, and we would like to define them in the beginning. The following will require the most basic of backgrounds. We will use n to denote the length of the strings in the various dimensions, or bases, which we discuss. The symbol $\circ$ will be used to indicate that the two objects on either side will be *concatenated* , or the first object will be added on the front of the second object. We use $\cup$ in the more traditional way, to mean the union of two sets of strings. The final symbol to define is the superscript $base^R$. This will denote the creating of a new set which consists of the base set in reverse order. With the symbolic explanations out of the way, we can introduce our main topic, error-correcting codes.

An error-correcting code on a graph consists of a subset of the vertices (these special vertices are called codewords) and a rule that given any vertex returns the codeword vertex nearest to the given vertex. Distance here is defined in the obvious way as the number of graph edges which are traversed on the shortest path between the two vertices. For this rule to make sense, there should be only one codeword which is closest to each vertex. When for some vertices there are two or more closest codewords, the code is said to detect an error, but it can't correct the error.

If each codeword is at distance 2d+1 from another codeword, then we can think of a codeword as sitting at the center of a 'sphere' of radius d. Each vertex in this sphere will be decoded as the codeword. When every vertex is in exactly one of these spheres, the code is a perfect d-error- correcting code. In particular, in a one-error-correcting code each codeword should be at distance 3 from another codeword. If the spheres of radius 1 around each codeword do not overlap and do include every vertex, then the code is a perfect one-error-correcting code. [1] In other words, every vertex is either a codeword, or shares an edge with a codeword. For our purposes, we would like the or to be an exclusive one.

The graphs must be labeled in a way that will result in easy error-correcting and decoding procedures. Given a string, we need a way to determine whether or not it is a codeword. This is done using certain recursive formulas and can be checked accordingly. For error-correcting, when given a non-codeword string, the procedure should send the string to the nearest codeword. The second procedure we need to examine for each possible labeling is decoding. For this, we want to input a string and output an integer which represents the position of the string in the graph (using standard ordering). These are the two essential procedures that we need to satisfy when developing labelings of the graphs.

Chapter 2

2.1 Labeling One-Dimensional Codes

The placement of codewords in the one-dimensional (linear) case is trivial with our specified restrictions. Using the two *rules* for vertex adjacency and the added restriction that we want zero, or the first vertex in the graph, to be a codeword, then the set of codewords is forced. It is obvious in this one-dimensional case that since the first vertex is always chosen as a codeword, every third vertex will then be a codeword.

The standard binary ordering can be used to label the one-dimensional graph. It has nice properties when applying easy error-correcting and decoding procedures. As it turns out, the codewords will be all strings (integers) congruent to 0 (mod 3). This results in the error- correcting procedure also being trivial. If a string is congruent to 1 (mod 3), then the codeword is immediately preceding the string. Similarly, if a string is congruent to 2 (mod 3), the codeword is the string immediately following the given string. With these amazingly simple results, one must question whether there exists any other labelings that give such easily definable error-correcting and decoding procedures.

We next examined a set of binary strings (but can also be used in other bases) known as Gray Codes, named for Frank Gray who worked as a researcher at Bell Laboratories' in the 1940's, who initially looked at these codes for analog to digital conversion, but since then have been applied in many other areas. These codes have the property that there is only one digit, or bit change between two consecutive Gray code strings. The most used of these codes is the standard binary reflective Gray code, which we will now define. Let G_n = the set of 2^n Gray code strings, each length n. We define this set with the following recursive formula:

$$G^n = 0 \circ G_{n-1} \cup 1 \circ G^R_{n-1}$$

where we define $G_1 = 0\ 1$. Looking at the two following sets, where n=2 and n=3, the Gray code string sets are given as follows:

$$G_2 = 00 \quad 01 \quad 11 \quad 10$$

$$G_3 = 000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100$$

Now we have another labeling for the one-dimensional graph which is easily defined. The next step was to examine its various properties in order to attempt a characterization of the codewords and to find possible error-correcting and decoding procedures.

2.2 Characterizing the Codewords of the Gray Code

The codewords for the Gray code labeling are not as easy to characterize as with the standard binary ordering, however, we used observations and *helper* strings to come up with a set of recursive formulas for our specific purpose. We noticed that when looking at the set of codewords for n even, it is merely the set of codewords for the previous n, with a few adjustments which seemed to conform to a pattern. So, our formula for C_n = the set of codewords of length n, which we will later prove, is given recursively by:

$$C_n = 0 \circ C_{n-1} \cup 1 \circ C^R_{n-1} \qquad n - even$$

The repetition of the Gray code recurrence relation showing up here is not surprising, as we shall later further develop the relationship.

The codewords for n odd, however, did not have such nice properties. But, further examination of the Gray code strings led to the creation of two *helper* strings. These strings are E_n = the set of all strings, length n, whose position is congruent to 1 (mod 3) and T_n = the set of all strings, length n, whose position is congruent to 2 (mod 3). These three sets allow us to characterize the codewords in a non-complex fashion. For example, with G_3 (shown above), the sets are:

$$C_3 = 000 \quad 010 \quad 101$$

$$E_3 = 001 \quad 110 \quad 100$$

$$T_3 = 011 \quad 111$$

The following six formulas completely characterize the codewords using the standard binary reflective Gray code labeling on the one-dimensional graph. The proof of these formulas follows, beginning with a lemma which is necessary to characterize the two different position strings on which each G_n ends.

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 1 \circ C^R_{n-1} \qquad n - even \\ E_n &= 0 \circ E_{n-1} \cup 1 \circ T^R_{n-1} \\ T_n &= 0 \circ T_{n-1} \cup 1 \circ E^R_{n-1} \end{aligned}$$

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 1 \circ E^R_{n-1} \qquad n - odd \\ E_n &= 0 \circ E_{n-1} \cup 1 \circ C^R_{n-1} \\ T_n &= 0 \circ T_{n-1} \cup 1 \circ T^R_{n-1} \end{aligned}$$

Initial conditions for these equations:

$$C_1 = 0$$

$$E_1 = 1$$

$$T_1 = \emptyset$$

$$T_2 = 11$$

First, we state the following lemma.

$$\textbf{Lemma}: \quad 2^n \equiv 1(mod3) \quad n-even \quad 2^n \equiv 2(mod3) \quad n-odd$$

Now that we have this lemma, we can prove that these equations do, in fact, characterize the codewords, C_n, and the two sets of strings, E_n and T_n, which represent the strings at positions $\equiv$ 1 and 2 (mod 3), respectively.

Proof: From the lemma and our restriction of the first vertex always being a codeword, we know what set the last string of each G_n will belong to. For n-even, the last string will be a codeword, or an element of C_n, and for n-odd, the last string will belong to E_n. We will now look at the case where n is even.

Let $C1_n$, $E1_n$, and $T1_n$ denote the first $half$ of strings in each respective set, which are formed, using the standard binary reflective Gray code formula, by concatenating a zero with the same set of C_{n-1}, E_{n-1}, or T_{n-1}, respectively. Let $C2_n$, $E2_n$, and $T2_n$ denote the second $half$ of strings in each respective set, which are formed by one concatenated with the same set of C_{n-1}, E_{n-1}, or T_{n-1}, respectively, in reverse order. It is evident that $C1_n$, $E1_n$, and $T1_n$ are trivial, due to the way in which G_n is formed. The strings will remain in the same order, and will only differ by a leading zero. Therefore, we only need to show that the second half of each of the six formulas (above) holds for $C2_n$, $E2_n$, and $T2_n$.

Since we are examining the n-even case, and the formulas are comprised of two (n-1) cases each, it is logical to first look at the (n-1) case. This case will, naturally, result in n being odd. Therefore, the last string in $E1_n$ will now be the first string in $T2_n$, since the Gray code merely copies the set of strings in reverse order. Likewise, the first string in $C2_n$ will be the last string from $C1_n$. It also follows that the first string in $E2_n$ will be the last string from $T1_n$. These all follow directly from the reflective property of the Gray code's recurrence relation. The proof for the n-odd case is omitted since it is similar to the even case and follows directly from the lemma and the recursive formula of the Gray Code.

2.3 Error-correcting and Decoding with the Gray Code Labeling

Now that we have a good way to characterize the codewords, we need to examine possible error-correcting and decoding procedures. The decoding procedure can be done easily by converting the Gray Code string into its binary counterpart. A Gray to binary conversion algorithm has been developed [2]. This conversion results in the position of the string on the graph being easily discovered, hence a simple decoding procedure. For error-correcting, we use the three sets of position strings since the given string will belong to one of these. Since there are three different positions in which the string could be, we need to examine three different cases. The trivial case is when the string is a codeword because the string is already at the nearest codeword. For the other two cases, we will need some way of "moving" the string to the nearest codeword. We discovered much research has been done on the Gray codes and, in addition to Gray to binary conversion, there are also two algorithms for incrementing and decrementing procedures for Gray code strings. So now, if the string is in E_n, we decrement the string and if it is in T_n, we increment the string. The two algorithms for incrementing and decrementing the Gray code strings are given by [2]. Therefore, the Gray code gives us a second way of labeling the one- dimensional graph which has nice error-correcting and decoding procedures. Our next logical step was to try and find other labelings of the one- dimensional graph which would produce similar results.

2.4 Generating Other Labelings

In order to attempt to generate other possible labelings, we looked at simulating a six finite-state machine. This can be thought of as six different states (or circles, pictorially), we will call them Even0, Even1, Even2, Odd0, Odd1, Odd2. Each state is pointing to two of the other states, which aren't necessarily different, and each are being pointed to by two states, which also aren't necessarily different. These input and output arrows represent the two different digits. We divide the six states into two groups of three which correspond to even and odd cases because of the cases for n both even and odd. The purpose of these machines is to match each binary string with its corresponding set of C_n, E_n, or T_n, so that we know which strings are in these sets. We can then attempt to match a labeling to the graph keeping the strings in their respective sets, while having the freedom to move the string within the set itself. To classify the binary string, we start at the rightmost digit and work our way to the left. The check is always started at the Even0 state and we follow the string, digit by digit, until its termination. Whichever state the string terminates in (0, 1, or 2) will correspond to one of the three sets C_n, E_n, or T_n, respectively. At the termination of each complete set of 2_n strings, we now know the make up of the three sets and can examine them for a possible labeling of the graph.

One problem which arises is the number of possible six state machines. Since the only restriction, so far, is that each one has two inputs and two outputs, that results in a tremendous number of possibilities. However, there are some properties that allow us to further restrict the number of machines that will work. Namely, we want Even0 to go to Odd0 if the digit is 0, and to go to Odd1 if the digit is 1. Also, we want the Odd0 to go to Even0 if the digit is a 0, since we presumably want 0 to be a codeword for all n. The Even0 to Odd1 case is again, since we presumably want 1 to go to the set E_n for all n. The other restrictions placed on the machines are due to the set number of strings needed in each set.

We implemented a simulation of the six state machines using a computer program (pp. 12-17) to try and generate. We did generate 48 machines whose counts for n=3,4,5 matched the counts of the number of strings needed in each set. Out of these 48, we thought that since the sensible way to begin the labeling was 0 and 1 as the first two strings, we looked at the 8 machines which conformed to this restriction for n=3. Out of these 8, there were 4 in which the strings 0 and 1 stayed in their respective positions for n=4,5. These machines give us which strings belong in which position strings, however, it does not give the order in which these strings should label the graph. The following table gives the machines on the left, the strings for each position set on the right, and the recursive relationships, in the same order which is indicated by the number to the upper left, on the next page, which hold through n=5. The program immediately follows these two tables.

Let E = even and D = odd.

MACHINE:

States:	**Input (0 or 1):**		**SETS:**
1			
Even0	**D0**	**D1**	C_3 = **(000,010,011)**
Even1	**D1**	**D2**	E_3 = **(001,100,111)**
Even2	**D0**	**D2**	T_3 = **(101,110)**
Odd0	**E0**	**E2**	C_4 = **(0000,0010,0011,1001,1100,1111)**
Odd1	**E1**	**E0**	E_4 = **(0001,0100,0101,0110,0111)**
Odd2	**E1**	**E2**	T_4 = **(1000,1010,1011,1101,1110)**
2			
Even0	**D0**	**D1**	C_3 = **(000,010,011)**
Even1	**D1**	**D2**	E_3 = **(001,100,111)**
Even2	**D0**	**D2**	T_3 = **(101,110)**
Odd0	**E0**	**E2**	C_4 = **(0000,0010,0011,1001,1100,1111)**
Odd1	**E1**	**E0**	E_4 = **(0001,0100,0111,1101,1110)**
Odd2	**E2**	**E1**	T_4 = **(0101,0110,1000,1010,1011)**
3			
Even0	**D0**	**D1**	C_3 = **(000,011,110)**
Even1	**D1**	**D2**	E_3 = **(001,100,111)**
Even2	**D2**	**D0**	T_3 = **(010,101)**
Odd0	**E0**	**E2**	C_4 = **(0000,0011,0110,1001,1100,1111)**
Odd1	**E1**	**E0**	E_4 = **(0001,0010,0100,0101,0111)**
Odd2	**E1**	**E2**	T_4 = **(1000,1010,1011,1101,1110)**
4			
Even0	**D0**	**D1**	C_3 = **(000,011,110)**
Even1	**D1**	**D2**	E_3 = **(001,100,111)**
Even2	**D2**	**D0**	T_3 = **(010,101)**
Odd0	**E0**	**E2**	C_4 = **(0000,0010,0011,1001,1100,1111)**
Odd1	**E1**	**E0**	E_4 = **(0001,0100,0111,1010,1101)**
Odd2	**E2**	**E1**	T_4 = **(0010,0101,1000,1011,1110)**

1

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 1 \circ E_{n-1} \qquad n - even \\ E_n &= 0 \circ E_{n-1} \cup 0 \circ T_{n-1} \\ T_n &= 1 \circ C_{n-1} \cup 1 \circ T_{n-1} \end{aligned}$$

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 0 \circ T_{n-1} \qquad n - odd \\ E_n &= 0 \circ E_{n-1} \cup 1 \circ C_{n-1} \\ T_n &= 1 \circ E_{n-1} \cup 1 \circ T_{n-1} \end{aligned}$$

2

$$\begin{aligned} C_n &= 1 \circ C_{n-1} \cup 1 \circ E_{n-1} \qquad n - even \\ E_n &= 0 \circ E_{n-1} \cup 1 \circ T_{n-1} \\ T_n &= 0 \circ T_{n-1} \cup 1 \circ C_{n-1} \end{aligned}$$

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 0 \circ T_{n-1} \qquad n - odd \\ E_n &= 0 \circ E_{n-1} \cup 1 \circ C_{n-1} \\ T_n &= 1 \circ E_{n-1} \cup 1 \circ T_{n-1} \end{aligned}$$

3

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 1 \circ E_{n-1} \qquad n - even \\ E_n &= 0 \circ E_{n-1} \cup 0 \circ T_{n-1} \\ T_n &= 1 \circ C_{n-1} \cup 1 \circ T_{n-1} \end{aligned}$$

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 1 \circ T_{n-1} \qquad n - odd \\ E_n &= 0 \circ E_{n-1} \cup 1 \circ C_{n-1} \\ T_n &= 0 \circ T_{n-1} \cup 1 \circ E_{n-1} \end{aligned}$$

4

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 1 \circ E_{n-1} \qquad n - even \\ E_n &= 0 \circ E_{n-1} \cup 1 \circ T_{n-1} \\ T_n &= 0 \circ T_{n-1} \cup 1 \circ C_{n-1} \end{aligned}$$

$$\begin{aligned} C_n &= 0 \circ C_{n-1} \cup 1 \circ T_{n-1} \qquad n - odd \\ E_n &= 0 \circ E_{n-1} \cup 1 \circ C_{n-1} \\ T_n &= 0 \circ T_{n-1} \cup 1 \circ E_{n-1} \end{aligned}$$

Initial conditions for these equations:

$$C_1 = 0$$

$$E_1 = 1$$

$$T_1 = \emptyset$$

$$T_2 = 10$$

```
//  Program:  FSM.c++
//  Written by:  Jessica Cavanaugh
//  Date started:  July 29, 1997 Date updated:  August 12, 1997
//
//  Purpose:  This program is designed to generate most possibilities of
//            labeling the one-dimensional one error-correcting code
//            by simulating six finite state machines.   These machines
//            correspond to a two-dimensional array, with the six rows
//            representing the states, and the two columns representing
//            the 0 and 1 digits, or inputs.  The object is to input
//            2^n strings of length n and test them in each of 64
//            possible machines.  By testing all of the strings, each
//            digit is looked at, in turn, starting at the righthand
//            most digit.  Using the two-dimensional array, each string
//            is followed, digit by digit, until its termination.  At
//            that point, the state in which it terminates is kept track
//            of and counted.  These counts are then checked after the
//            2^n strings are put through each machine.  Since it is
//            known what the counts must be for each n, we are looking
//            for the machines that match these counts.

#include <fstream.h>
#include <iostream.h>
#include <math.h>

//  *****************Variable Declarations**************************

int size,                          // string size and # of strings

    s,r,p,v,j,x,y,z,f,g,i,k,l,m,h,b,            // loop variables

    C1,C2,C3,                          // counting the termination of
      // the strings at positions 0,1,2

    digit,place,temp,unit,nmber,num;       // used to test the strings

int E1,E2,D1,D2;                   // hold the values that the counts
```

```
                      // are compared to

int  w,n,nmb,hold,expt;               // used to test the strings

int C[6][2];                    // the 2-D array holding the machines

int count[6];                      // holds the count of each state

//  ************************Main Program**************************

main ()
{ ofstream outdata;           // creating a file to hold output
  outdata.open("out.dat");
 size=2;
 n=2;
 for (r=0;r<(n-1);r++)
  {
     size = size*2;  }
 i=1;
int  S[size];
 S[0]=0;
 while (i<size)
 {   w=(n-1);
     S[i]=0;
     p=i;                               // This section fills the
     while (p>0)                       // array of binary strings with
     {  unit=1;                        // 2^n strings of length n
nmber=1;
        f=0;
        while (f<w)
   {  unit=unit*2;
      f++;   }
if (p>=unit)
  {   g=0;
      while (g<w)
 { nmber=nmber*10;
   g++;   }
```

```
      S[i]=S[i]+nmber;
      p=p-unit;
      w=w-1;
  }
       else   w=w-1;
     }
i=i+1;
}

C[0][0]=3;                  // Initializing the Even 0 state and the 0
C[0][1]=4;                  // digit of the Odd 0 state, to insure that
C[3][0]=0;                  // the codewords end in the correct states

D1=(size+1)/3;
D2=(size-2)/3;              // Initializing the values that the count
E1=(size+2)/3;              // should be at each state; depends on n
E2=(size-1)/3;

for (i=0;i<4;i++)
  { if (i==0)
      {  C[1][0]=4;            // First for loop; initializes the Even 1
         C[1][1]=5;  }        // state with 4 different possibilities
    else if (i==1)
   {  C[1][0]=5;
      C[0][1]=4;  }
         else if (i==2)
{  C[1][0]=3;
   C[1][1]=5;  }
              else { C[1][0]=5;
     C[1][1]=3;  }

   for (j=0;j<2;j++)         // Second for loop; initializes the Even 2
     { if (i>=2)            // state with 4 different possibilities
{ if (j==0)
   {  C[2][0]=4;
      C[2][1]=5;  }
         else {  C[2][0]=5;
```

```
 C[2][1]=4;  } }
      else if (j==0)
     {  C[2][0]=3;
        C[2][1]=5;  }
           else {  C[2][0]=5;
   C[2][1]=3;  }

      for (k=0;k<2;k++)       // Third for loop; initializes the Odd 2
 {  if (k==0)
    {  C[5][0]=1;       // state with 2 different possibilities
       C[5][1]=2;  }
          else {  C[5][0]=2;
  C[5][1]=1;  }

 for (l=0;l<2;l++)     // Fourth for loop; initializes the
   { if (l==0)       // Odd 0 state, 1 digit; 2 possibilities
       { C[3][1]=1; }
             else { C[3][1]=2; }

       for (m=0;m<2;m++)    // Fifth for loop; initializes the
 { if (l==0)       // Odd 1 state with 4 possibilities
     { if (m==0)
 {  C[4][0]=2;
    C[4][1]=0;  }
                       else { C[4][0]=0;
      C[4][1]=2;  } }
                  else { if (m==0)
   {  C[4][0]=1;
      C[4][1]=0;  }
                         else {  C[4][0]=0;
 C[4][1]=1;  } }

for (v=0;v<6;v++)
count[v]=0;

for (y=0;y<size;y++)
{
```

```
place=0;
for (x=0;x<n;x++)
 {     hold=1;
               h=0;
       while (h<x)
   {  hold=hold*10;
      h++;         }      // this section takes each
       num=(S[y]/hold);             // n length string and
       digit=num%2;              // runs it through one
       temp=C[place][digit];       // machine at a time,
               if (x==(n-1))               // string ends
       {  count[temp]++;
          outdata << temp << endl;
       }
               place=temp;

         }  // end for(x)
if (y==(size-1))
{ outdata << endl;
     C1=count[0]+count[3];   // this section prints out the
     C2=count[1]+count[4];   // machine if it matches the
     C3=count[2]+count[5];   // desired counts
             if ((n%2)==0)
       { if ((C1==E1)&&(C2==E2)&&(C3==E2))
   { //for (z=0;z<6;z++)
//    { outdata << C[z][0] << "  " << C[z][1];
//      outdata << endl;  }
} }
             else { if ((C1==D1)&&(C2==D1)&&(C3==D2))
                     {// for (b=0;b<6;b++)
 //     {  outdata << C[b][0] << "  " << C[b][1];
//  outdata << endl;  }
 } }
} //end if
} //end for(y)
outdata << endl;
      } //end for(m)
```

```
  } //end for(l)
 } //end for(k)
} //end for(j)
} //end for(i)
outdata.close();
return 0;
} // end main()
```

It is noted that the Gray code labeling did not come up with the output. We realized that the program missed some possible machines by being to restrictive, but unfortunately it was too late in the summer to make the necessary adjustments. Therefore, we still do not know if there are other possible labelings of the one-dimensional graph. However, our search for error-correcting codes does not end with one-dimensional graphs. The latter half of the paper will deal briefly with two-dimensional graphs, for background purposes mostly, and largely with three-dimensional graphs. Labelings, placement of codewords, and characterizations of the codewords in three dimensions will all be examined and generalized to n dimensions as well.

Chapter 3

Unlabelled graph topology

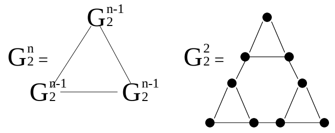
3.1 Uncoded topology in N dimensions

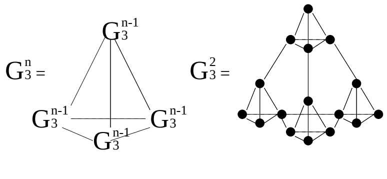
Ultimately, we will be dealing with graphs in N dimensions labelled with strings in base $N+1$. These graphs belong to a general family which has, for a level n (corresponding to the length of the strings necessary to label all of the graph's points), $(N+1)^n$ vertices. Each interior, non-corner vertex is connected to $N+1$ other vertices. There are $N+1$ corner vertices which are connected to N other points.

In all dimensions, graphs of this type are easily described recursively. In one dimension the unlabelled graph G_1^n is defined by the recursion:

$$G_1^n = G_1^{n-1} \text{---} G_1^{n-1}$$

The two dimenional and three dimensional cases are described by the following figures:





In all cases, G_N^0 is a single vertex.

3.2 The coded graph in one and two dimensions

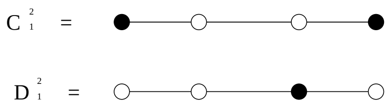
It becomes increasingly complicated to precisely define the structure of a perfect-one-error-correcting code (p-1-ecc), the **coded graph**, for higher dimensions. In one dimension, the p-1-ecc seems transparently obvious given the first vertex as a code point. Its uniqueness is intuitive; every third point must be defined as a code vertex. Let C_N^n be the coded graph in dimension N. The one dimensional structure can be defined by the following diagram:

$$C_1^n = C_1^{n-1} \text{—} {C_1^{n-1}}^R \quad n \text{ even}$$

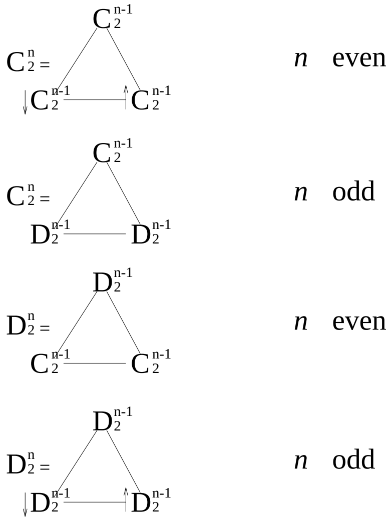
$$C_1^n = C_1^{n-1} \text{—} D_1^{n-1} \quad n \text{ odd}$$

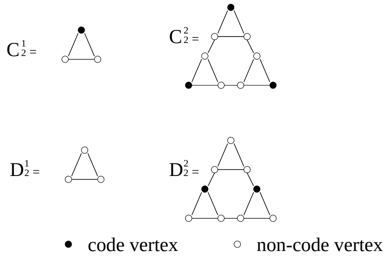
$$D_1^n = D_1^{n-1} \text{—} C_1^{n-1} \quad n \text{ even}$$

$$D_1^n = D_1^{n-1} \text{—} {D_1^{n-1}}^R \quad n \text{ odd}$$



C_1^{nR} is defined as the reflection of C_1^n. Cull and Nelson [1] proved that the coded topology of the two dimenional graphs can be constructed as follows:

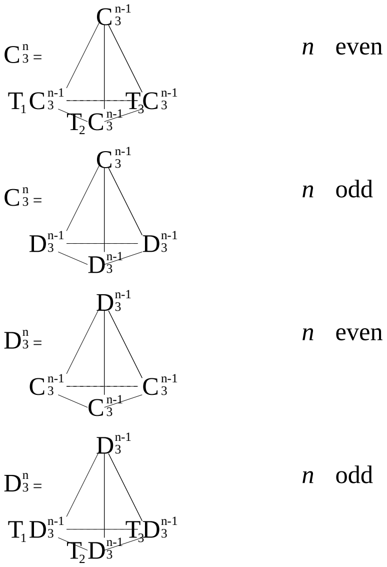


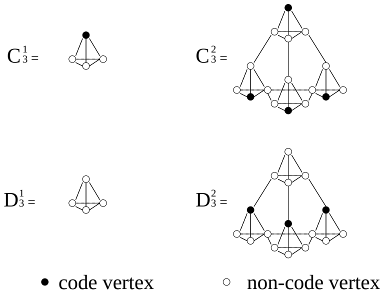


$\downarrow C_2^n$ is a copy of C_2^n rotated 120 degrees counter-clockwise and $\uparrow C_2^n$ is a copy rotated 120 degress clockwise. In both cases C_N^0 is a codepoint and D_N^0 is not.

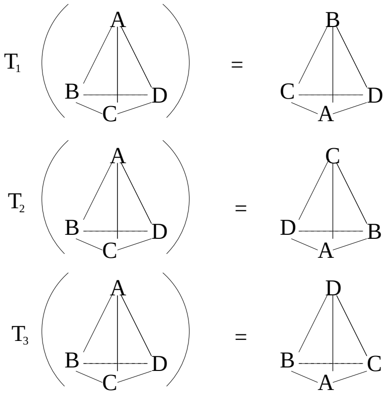
3.3 The coded graph in dimensions $N \geq 3$

In three dimensions, we propose a coded graph involving a more complicated symmetry in its recursive construction. Of course, C_3^n has the same topology as G_3^n. The following recursive diagrams describe the creation of the C_3^n coded graphs:





Again, C_3^0 is a codepoint and D_3^0 is not a codepoint. The T transformations rotate a copy of the previous graph so that the top corner point of the graph is moved to the middle-lower corner point and one of the other corners; different for T_1, T_2 and T_3; is moved to the top corner position. In other words, if A,B,C and D represent the corner points of the C_3^n graph:



We note that the recursive constructions follow a pattern in dimensions one,two and three. In the even n cases, C_N^n is constructed from a copy of C_N^{n-1} connected to N copies of itself. Without describing the precise symmetry, the N lower copies of C_N^{n-1} are all maneuvered so that the corner A in C_N^{n-1} becomes the corners of the graph C_N^n. In the odd n cases, C_N^n is created by attaching N copies of a helper graph D_N^{n-1} to the lower corners of C_N^{n-1}. The helper graphs are constructed similarly to the code graphs with the odd and even cases reversed. Graphs with these characteristics could be constructed in all dimensions.

Theorem 3.1 *For all $n \geq 0$, C_N^n defines a perfect-one-error- correcting code.*

proof. The construction can be followed to demonstrate that C_N^0, C_N^1 and C_N^2 define p-1-ecc's for $N = 0, 1, 2$. C_N^0 and C_N^1 clearly hold through the hypotheses for all N. For $n = 2$, we could argue that, as the only code vertex in C_N^1 is the A corner and this has been moved to all corners in C_N^2, all inner connections between the component C_N^1 graphs are between non-code points–the non-code corners of C_N^1. The construction connects each point to one and only one code point and no two code points are adjacent.

Recall that a graph defines a p-1-ecc if and only if each non-code vertex is at distance one from exactly one code vertex and no code points are adjacent.

Case 1: n even.

Assume C_N^{n-1} defines a p-1-ecc. As C_N^n is comprised of $N+1$ components which define p-1-ecc's, the only potential problems in the C_N^n graph are at the component connection points. By the transformations, each component graph will have connections at all of its corners except A. Therefore, if all corners except A were non-code vertices in C_N^{n-1}, the connection question would be resolved. As $n-1$ is odd, if no corner vertices in D_N^{n-2} are code points, lower corners in C_N^{n-1} would be non-code. However, as $n-2$ is even, all corners in D_N^{n-2} are non-code if and only if the lower corners in C_N^{n-3} are non-code. It is clear from the construction that lower corners in C_N^1 are non-code. Assuming non-code lower vertices in C_N^{n-3} implies non-code lower vertices in C_N^{n-1}. By induction, all lower vertices are non-code in C_N^{n-1} for even n. This implies C_N^n defines a p-1-ecc.

Case 2: n odd.

The following multi-part lemma will be necessary to prove this case.

Lemma 3.1 *In D_N^n for $n \geq 2$:*
a) Each non-corner, non-code vertex is a distance one from exactly one code vertex. No code vertices are adjacent.
b) If n is odd all corner vertices are distance two from exactly one code vertex.
c) If n is even the A corner is distance two from exactly one code vertex and all other corners are distance one from exactly one code point.

proof. These conditions are easily demonstrated by a constructive argument in D_N^2 and D_N^3.

Case I: n odd.

It was demonstrated above that, for even n, D_N^n had all non-code corners. By the construction, this immediately implies non-code corners in the D_N^n n odd case as all corners in D_N^n were the A corner in D_N^{n-1}. From this, we can see that if condition (b) holds for the A corner in D_N^{n-1} it will hold for all corners in D_N^n. In other words, the first part of (c) implies (b). By construction, the A vertex is unchanged for all n. Therefore, as (c) holds for D_N^2, it holds for all D_N^n. This implies (b).

Case II: n even.

The first part of (c) was demonstrated above. The lower corners of D_N^n are, by the construction, lower corners of the graph C_N^{n-1} where $n-1$ is odd. It was argued above in the case of C_N^n for n even that the lower corners of C_N^{n-1} were non-code. As stated above, demonstration shows that (c) holds for D_N^2. Assuming (c) in D_N^{n-2} implies the lower corners of C_N^{n-1} are distance one from one and only one code vertex and this directly implies (c) for D_N^n. Therefore, by induction, (c) holds for D_N^n for all even n.

Condition (a) clearly holds, by a simple inductive argument, at all vertices with the possible problem of the connections between components. For n even we know by condition (b) that lower corners in D_N^{n-1} are distance two from any code vertex. In D_N^n these are connected to the A corner in C_N^{n-1} which is a code vertex by definition. Thus, the lower corners in D_N^{n-1} are a distance one from exactly one code point and (a) holds. For n odd, condition (c) allows virtually the same argument to be made as in the even case. Connections occur at lower corners of D_N^{n-1} which are non-code but are distance one from exactly one code vertex within their substructure. After the construction they remain distance one from precisely one code point.

Finally, Case 2 can be proved. By (a), C_N^n defines a p-1-ecc at all points with the possible exception of connection points and corners. Corner A is a code vertex for all n. The lower corners are lower corners of the D_N^{n-1} graphs which, by (c) are distance one from exactly one code point. By hypothesis, the lower corners of C_N^{n-1} are the A corner of C_N^{n-2}–always a code vertex.

These are connected to A corners in D_N^{n-1} which, by (c) are distance two from any code points. Therefore, these connections points are distance one from only one code vertex. All potential problems check out. Thus C_N^n defines a p-1-ecc. □

3.4 Uniqueness of the coded graph

We have demonstrated that the coding structure described above describes a p-1-ecc in all dimensions on this family of graphs. However, many graphs have a multiplicity of potential p-1-ecc's. We will now prove that the p-1-ecc defined above is the only possible perfect code with the top vertex chosen a code point on this family of graphs. We will expand the rather complicated proof Cull and Nelson [1] used to prove the uniqueness of their code structure on the two dimensional graph. A **consistent 3-labelling** is an assignment of a label from $\{C, B, R\}$ to each vertex so that the following restrictions are met:

a) Each component $n = 1$ structure is labelled in one of the four following manners: if C is the apex vertex, all lower vertices are labelled R; if R is the apex vertex, all lower vertices are labelled B; and if B is the apex vertex, one of the lower vertices is labelled C and the others are all labelled R.

b) The labelling of a *successor* of a vertex satisfies $succ(C) = R, succ(R) = B, succ(B) \in \{R, C\}$.

c) No two vertices labelled C are adjacent.

d) Vertices labelled B in the bottom corners of the graph are not adjacent to a C, but every other B is adjacent to a C.

e) If the top vertex is labelled R then it is not adjacent to a C but all other R's are adjacent to exactly one C.

Lemma 3.2 *A p-1-ecc of G_N^n induces a consistent 3-labelling of G_N^n.*

Proof.: Given the p-1-ecc, label each codeword C, label each successor of a codeword R and label each predecessor of a codeword B. Now one can label the remaining vertices by labeling the successors of R's with B and the successors of B's with R. As all vertices but the top corner have exactly one predecessor, all vertices will now be labelled. Consider the top corner. Either it is a codeword and is, therefore, labelled C, or, by the P-1-ecc assumption, it is the predecessor of a codeword and is hence labelled B. Now we must check that the restrictions from above are all satisfied. For condition (a), consider the top vertex of each component $n = 1$ structure. If the top vertex is labelled C then the lower vertices, as successors of a codeword, are all labelled R giving the correct labelling. If the top vertex is labelled R, then none of its successors is a C because this would imply that the top vertex was labelled B, not R. None of the lower vertices are labelled R as none of them are successors of a code vertex or of a B. Therefore, all lower vertices must be labelled B, and this satisfies (a). Lastly, if the top vertex is a B, as a predecessor of a codeword, one of the lower vertices is labelled C. By hypothesis, the graph has a p-1-ecc, therefore the C vertex cannot be adjacent to any other C vertices. By construction, all other successors of a B are labelled R. Again, condition (a) is met. Condition (b) follows directly from the construction of the labelling. Condition (c) comes from the fact that the graph has a p-1-ecc. Restrictions (d) and (e) follow because in a p-1-ecc each non-code vertex is adjacent to exactly one code vertex. Therefore, each B and R is adjacent to exactly one C. The exceptions in (d) and (e) cannot arise. As shown above, the top vertex of the graph cannot be labelled R. A bottom corner cannot be labelled B because, by (a), it would not be connected to any code vertex and would violate the p-1-ecc.□

Now we show that a consistent 3-labelling can only be erected from smaller consistent 3-labellings.

Lemma 3.3 *Any consistent 3-labelling of G_N^n gives a consistent 3-labelling for each of the G_N^n's constituent G_N^{n-1}*

Proof.: Since G_N^n is constructed from $N + 1$ copies of G_N^{n-1} connected as described above, the restrictions (a), (b) and (c) for G_N^n imply the same restrictions on the G_N^{n-1}'s. For (d), if any of the $N + 1$ G_N^{n-1}'s has a B in the bottom corner, then by (a) in the $n = 1$ figure that this corner is a part of, the B is only connected to other B's in the lower vertices and the top vertex is a R. It would not, therefore, be adjacent to any C is the G_N^{n-1}. All other B's are not corner vertices of either G_N^n or G_N^{n-1} and by (d) for G_N^n, these B's are adjacent to a C. For (e), if any R is a top vertex of an G_N^{n-1} then it is either the top vertex of G_N^n and, by (e) for G_N^n is not adjacent to any C, or it is internal for G_N^n and is, therefore, adjacent to exactly one C. By (a), though, this R is the apex of a $n = 1$ structure with all B's in the lower vertices and is thus not adjacent to a C in the G_N^{n-1}. All other R's are adjacent to exactly one C by (e) for G_N^n.□

Next we will show that only a few consistent 3-labellings of G_N^n actually exist.

Lemma 3.4 *For each $n \geq 1$, there are exactly three possible consistent 3-labellings of G_N^n and they have the following forms: for odd n, the corner vertices have one of the following three patterns:*
(i) C is the top corner and the lower corners are all R's,
(ii) R is the top corner and the lower corners are all B's,
(iii)B is the top corner, one of the lower corners is a C and all other lower corners are R's.
For even n, the corner vertices have one of the following three patterns:
(iv) all corners are C's,
(v) all corners are R's,
(vi) the top corner is a B, one lower corner is a B and all other lower corners are R's.

Proof.: For $n = 1$, G_N^n is a single component structure and by (a) in the definition, the only possible consistent 3-labellings are case (i), (ii) and (iii). Obviously, all definition conditions are met by these labellings. For $n > 1$, the previous lemma states that consistent 3-labellings can only be built from smaller consistent 3-labellings. In particular, for even n the consistent 3-labelling can only be built from the consistent 3-labellings of the odd n. Starting with (i) for the top G_N^{n-1} in G_N^n, the successor restrictions require the lower G_N^{n-1}'s to be labelled with the possible (iii) patterns. Conditions (c) and (e) in the definitions and by the constructive nature of this family of graphs, the C labelled lower vertex in the lower G_N^{n-1}'s must be a lower corner in G_N^n. This labelling satisfies all five definition conditions and corresponds to the even n labelling possibility (iv). If the top vertex is R, the top G_N^{n-1} has pattern (ii). By (d), the bottom B's are not adjacent to a C. To satisfy (d) for G_N^n, the lower G_N^{n-1}'s must have pattern (i) yielding a G_N^n which inherits all of the definition requirements from G_N^{n-1}, and corresponds to corner labelling (v). Similarly, if the top vertex of G_N^n with n even is labelled B, then the top G_N^{n-1} is labelled in pattern (iii). The bottom G_N^{n-1}'s can have the forms (ii) or (iii). However, by (d), the $B's$ from form (ii) which

get connected to another components must be connected to a C. Thus the possible labellings are restricted to having form (ii) connected to the C corner in the top G_N^{n-1} and the other lower labellings are all of the type (iii) with the C corners all attached to the type (ii) labelled G_N^{n-1}. These labellings will inherit the definition conditions from those properties for the G_N^{n-1}'s and have the type (vi) labellings in the listed possibilities. Now, for n odd and $n > 1$, we must start with one of the three even patterns for G_N^{n-1} and build the possible labellings for G_N^n. Using the successor rules, the fact that if a B in a bottom corner is connected to another vertex it must be connected to a C and an R cannot be connected to a C, there are only three possibilities:
1) If (iv) is the top labelling, the lower labellings must be (v)'s forming labelling (i).
2) If (v) is the top labelling, the bottom labellings must be of type (vi) with the B's on the corners of G_N^n forming the labelling (ii).
3) If (vi) is the top labelling, the B corner must be attached to a labelling (iv) and all top R labellings must attach to labelling (vi)'s with the B's connected to the type (iv) labelling. This corresponds to labelling (iii). It is a simple matter to check that each of these resultant G_N^n's inherit the definition restrictions for those properties in the G_N^{n-1}'s. Note also that there is exactly one consistent 3-labelling for each of these patterns. Thus the lemma is established. □

Finally, we are in a position to prove the uniqueness theorem for the coding structure in all dimensions of this family of graphs.

Theorem 3.2 *For each $n \geq 0$, there is a strictly unique perfect-1-error-correcting code for he graph G_N^n if the top vertex is required to be a code vertex.*

Proof.: For $n = 0$, G_N^n is a single vertex and when this point is a code vertex it is a unique p-1-ecc. For $n \geq 1$, by lemma if there is a p-1-ecc there must be a consistent 3-labelling. By lemma there are three such labellings for each n. For even n, labellings (v) and (vi) cannot give p-1-ecc's because they either contain a top R or a bottom B which is not adjacent to a C. The P-1-ecc is formed by making every vertex labelled C a codeword and each vertex labelled R or B a non-codeword. Therefore, for even n, the only pattern which gives a p-1-ecc is (iv). For odd n, one of the three labellings contains a bottom B and so cannot give a p-1-ecc. There is only one labelling which has a C at the top (i) and so this is unique. Thus, by recursive construction, the p-1-ecc is unique for all n. □

We have, therefore, defined the unique p-1-ecc for this family of graphs in all dimensions.

3.5 Code vertex count for all n

Having described a perfect-one-error correction code recursively for all dimensions N, calculating the number of code vertices for any n is a relatively simple matter of solving difference equations. Let c_N^n be the number of code vertices in C_N^n and d_N^n be the number in D_N^n. The following difference equations are derived immediately from the recursive construction:

$$\begin{gathered}
c_N^n = (N+1)c_N^{n-1} \quad n \text{ even} \\
c_N^n = c_N^{n-1} + (N)d_N^{n-1} \quad n \text{ odd} \\
d_N^n = d_N^{n-1} + (N)c_N^{n-1} \quad n \text{ even} \\
d_N^n = (N+1)d_N^{n-1} \quad n \text{ odd} \\
c_N^0 = 1 \qquad d_N^0 = 0
\end{gathered}$$

Observing that $c_N^n - d_N^n = c_N^{n-1} - d_N^{n-1} = 1$ for all n, it follows that $c_N^n = (N+1)c_N^{n-1} - (N)|\sin\frac{(n\pi)}{2}|$ for all n. The solution to the equations is presented in the following lemma:

Lemma 3.5 *The number of code vertices in C_N^n is:*

$$\frac{(N+1)^n + (N+1)}{N+2} \quad n \textit{ even}$$

$$\frac{(N+1)^n + 1}{N+2} \quad n \textit{ odd}$$

As the coding structure is unique for all N, this code vertex count is, of course, also unique for p-1-ecc's on this family of graphs.

Chapter 4

The labelled graph

4.1 The Towers of Hanoi labelling

Cull and Nelson [1] demonstrated how the Towers of Hanoi puzzle could be used to derive a simple and easily described labelling for the two dimensional graph. Code words, the words labelling code vertices, were easily described and decoding could be done simply with a finite state machine. The T of H labelling resembles the Gray code in its preservation of the rule that all words length one away from a word w differ from w by only one bit. It is perhaps also noteworthy that the Gray code can be used to solve the T of H puzzle as well.

When attempting to define a labelling for the three dimensional structure, we initially hoped to to find a way of again using the T of H puzzle in a slightly modified form. Ideally, the one change rule would be preserved. Unfortunately, our attempts yielded no result. The apparent lack of a means of using this puzzle to describe higher dimensional labellings may indicate that its remarkably strong association with the two dimensional coded graph is a fascinating coincidence.

4.2 Similarities between the Gray code and the T of H labellings

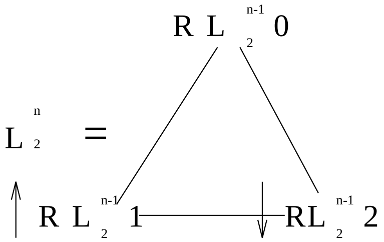
The recursive labelling formulas for both the Gray and T of H labellings also are both nearly identical to the recursive coded graph construction in the even case. The Gray code labelling is given by:

$$L_1^n = 0 \circ L_1^{n-1} \text{—} 1 \circ {L_1^{n-1}}^R$$

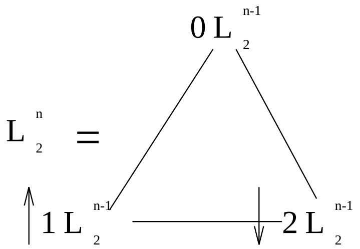
and recall that the coded graph structure for even n was:

$$L_1^n = C_1^{n-1} \text{—} {C_1^{n-1}}^R$$

The T of H labelling is described by:



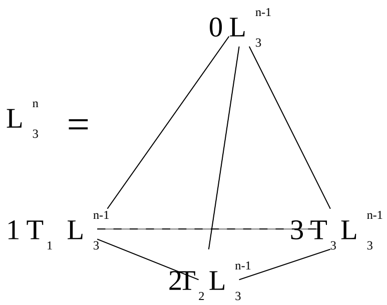
Without the reflections, the labelling:

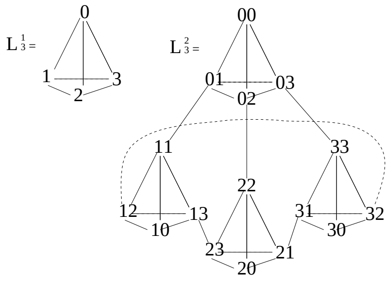


also yields a labelling which preserves the one change rule although the codewords do not appear to be as well defined as in the T of H labelling. Both, again, closely correspond to the even n case in the coded graph construction. However, it is noteworthy that the correlation is not as precise as in the one dimensional case.

4.3 Extending the pattern to three dimensions

Extending the general pattern exhibited in the Gray code type labellings in the first two dimensions to three dimensions, we hypothesize that a good labelling for the three dimensional graph would be given by the recursive structure:





Here, T_1, T_2, and T_3 are the transformations as defined above in the description of the unlabelled coded graph structure. This labelling follows the one change rule and yields code words which are easily derivable in the even n case.

The derivation of code words in the odd case is not nearly as clear upon examination through the first few n values. This labelling does not seem to offer the simple code word descriptions and decoding procedures of the T of H labelling. In the next section, we present hypotheses for the odd and even code word descriptions.

4.4 Defining the code words in L_3^n

In the case of n even, the labelling structure corresponds precisely to the construction of the coded graph. Thus it seems relatively clear that, if W_3^n is the set of code words:

$$W_3^n = 0 \circ W_3^{n-1} \cup 1 \circ W_3^{n-1} \cup 2 \circ W_3^{n-1} \cup 3 \circ W_3^{n-1} \quad n \text{ even}$$

In the n odd case, C_3^n is partially constructed by helper graphs. Therefore, W_3^n will likely be built from at least one helper set. However, the construction of such a set(s) does not appear to be a transparent problem. One possibility involves a rather complicated transformation of the helping sets X_3^n which corresponds rather loosely to the construction of the D_3^n in the coded topology.

Hypothesis:

$$W_3^n = 0 \circ W_3^{n-1} \cup 1 \circ t_{1 \ldots} X_3^{n-1} \cup 2 \circ t_{2 \ldots} X_3^{n-1} \cup 3 \circ t_{3 \ldots} X_3^{n-1} \quad n \text{ odd}$$

$$X_3^n = 0 \circ X_3^{n-1} \cup 1 \circ W_3^{n-1} \cup 2 \circ W_3^{n-1} \cup 3 \circ W_3^{n-1} \quad n \text{ even}$$

$$X_3^n = 0 \circ X_3^{n-1} \cup 1 \circ X_3^{n-1} \cup 2 \circ X_3^{n-1} \cup 3 \circ X_3^{n-1} \quad n \text{ odd}$$

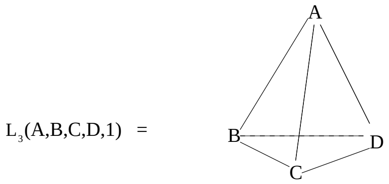
Here, t_1 takes $0 \leftrightarrow 1$ and $2 \leftrightarrow 3$, t_2 takes $0 \leftrightarrow 2$ and $1 \leftrightarrow 3$ and t_3 takes $0 \leftrightarrow 3$ and $1 \leftrightarrow 2$. The transformation $t_{1\ldots}$ implies that, for a string $abcde$, t_1 is applied to a. However, the order of operation on the string begins with the lowest order bit, the one furthest to the right. t_1 is applied to e if $d = 1$, t_2 if $d = 2$ and t_3 if $d = 3$. Similarly, d's transformation depends upon c. This procedure continues until t_1 is applied to a. $t_{2\ldots}$ and $t_{3\ldots}$ have identically correspondent defintions. This is a conjecture based solely upon its effectiveness through $n = 4$ and is, therefore, rather unsupported. A program generating the labelling for higher n would be necessary in order to better analyze the odd code word pattern.

4.5 A recursive algorithm to generate labellings

For any value of n, there is a simple means of producing the three dimensional labelling theorized above. The following formula can quickly produce the ordered labelling:

$$\begin{gathered} L_3(A,B,C,D,n) = A \circ L_3(A,B,C,D,n-1), B \circ L_3(B,C,A,D,n-1), \\ C \circ L_3(C,D,A,B,n-1), D \circ L_3(D,B,A,C,n-1) \quad \text{if } A = 0 \end{gathered}$$

$$\begin{gathered} L_3(A,B,C,D,n) = A \circ L_3(C,D,A,B,n-1), B \circ L_3(D,A,C,B,n-1), \\ C \circ L_3(A,B,C,D,n-1), D \circ L_3(B,D,C,A,n-1) \quad \text{if } A \neq 0 \end{gathered}$$



A, B, C, and D are initially all set to $0, 1, 2$ and 3 respectively but are reset for each new case. In the $A = 0$ case, the recursive formula comes directly from the transformations used in the labelled structure. In the $A \neq 0$ case, the formula corresponds to the inner transformations occuring in the lower, interior component graphs although the relationship is less obvious. This type of formula can also be used to generate the Gray code labelling in the one dimensional case:

$$L_2(A,B,n) = A \circ L_2(A,B,n-1), B \circ L_2(B,A,n-1) \quad \text{if } A = 0$$

$$L_2(A,B,n) = A \circ L_2(B,A,n-1), B \circ L_2(A,B,n-1) \quad \text{if } A \neq 0$$

Once again, the $A=0$ case's relationship to the labelled recursive graph is transparent. Here, though, the $A \neq 0$ (i.e. $A=1$) case's connection to the coded graph structure is easier to see. Using this relation, the standard binary positions of Gray code strings can be determined. The following example will illustrate the algorithm's implementation:

Example: What position is the Gray coded string 101101 in?
The string has $n=6$ digits. Therefore, it can be in positions 0 through $2^6-1=63$. The leading one indicates that the string lies in the second half of the graph by the $A=0$ case, in positions 32-63. This leading 1 places the string now in the $A \neq 0$ case and, as the next digit is $0=B$, the string is placed in the second half of the sub-graph, in positions 48-63. Still in the $A \neq 0$ case as A is again 1, the next 1 moves the potential positions in the 48-55 section. Now $A=0$, so the window is squeezed to 52-55 by the 1. That 1 causes the following 0 to position the string in 54-55 and the final 1 confirms the position at 54. This corresponds to the standard binary string 110110, a code word as 54 is 0(mod 3). Taking the labelling through its construction demonstrates that this is, indeed, the correct result.

Recalling that Gray code type labellings in two dimensions did not possess as close a relation to the coded structure as in the one dimensional, and apparently three dimensional, cases, it is not entirely surprising that there is not a recursive formula of this exact type for two dimensions. In order to attain a Gray code labelling, three different formula cases must be used, not the two cases of the one and three dimensional cases.

$$L_2(A,B,C,n) = A \circ L(A,B,C,n-1), B \circ L(B,C,A,n-1), C \circ L(C,A,B,n-1) \quad \text{if} A=0$$

$$L_2(A,B,C,n) = A \circ L(B,C,A,n-1), B \circ L(C,A,B,n-1), C \circ L(A,B,C,n-1) \quad \text{if} A=1$$

$$L_2(A,B,C,n) = A \circ L(C,A,B,n-1), B \circ L(A,B,C,n-1), C \circ L(B,C,A,n-1) \quad \text{if} A=2$$

Perhaps the difference is somehow related to the topology of the structure in the two dimensional case and this might be a commonality in all even dimensions. Even dimensions may require a three case recursive formula to arrive at a one-change labelling while odd dimensions only require two cases.

4.6 Standard labellings

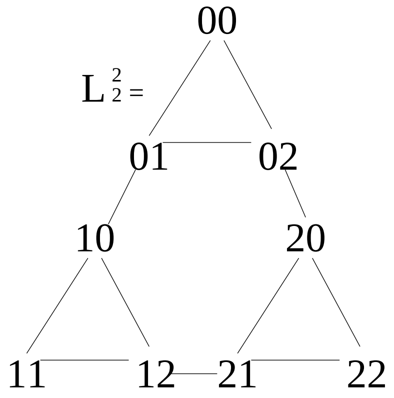
It is easy to see how a recursive formula of the form above can be used to give the words in the standard binary labelling in one dimension:

$$L_1(A,B,n) = A \circ L_1(A,B,n-1), B \circ L_1(A,B,n-1) \quad \text{for all } n$$

We conjecture that this could be expanded to higher dimensions in the same form to generate **standard** labellings for all N. The two and three dimensional expansions are given below:

$$L_2(A,B,C,n) = A{\circ}L_2(A,B,C,n{-}1), B{\circ}L_2(A,B,C,n{-}1), C{\circ}L_2(A,B,C,n{-}1)$$

$$L_3(A,B,C,D,n) = A \circ L_3(A,B,C,D,n-1), B \circ L_3(A,B,C,D,n-1),$$
$$C \circ L_3(A,B,C,D,n-1), D \circ L_3(A,B,C,D,n-1)$$



Standard labelling in two dimensions

Defining codewords in these higher dimensions is not as simple as in the binary case where codewords are simply multiples of three. The set of codewords in two dimensions is generated by the following recursion:

$$W_2^1 = 0 \qquad X_2^1 = \emptyset$$

$$W_2^n = 0 \circ W_2^{n-1} \cup 1 \circ \overline{W_2^{n-1}} \cup 2 \circ \overbrace{W_2^{n-1}} \quad n\text{even}$$

$$W_2^n = 0 \circ W_2^{n-1} \cup 1 \circ X_2^{n-1} \cup 2 \circ X_2^{n-1} \quad n\text{odd}$$

$$X_2^n = 0 \circ X_2^{n-1} \cup 1 \circ W_2^{n-1} \cup 2 \circ W_2^{n-1} \quad n\text{even}$$

$$X_2^n = 0 \circ X_2^{n-1} \cup 1 \circ \overline{X_2^{n-1}} \cup 2 \circ \overbrace{X_2^{n-1}} \quad n\text{odd}$$

Where $0 \circ \emptyset = \emptyset, \overline{W_2^{n-1}} = W_2^{n-1}$ with 0's and 1's interchanged $(0 \leftrightarrow 1)$ and $\overbrace{W_2^{n-1}} = W_2^{n-1}$ with 0's and 2's interchanged $(0 \leftrightarrow 2)$. Solving the difference equations generated by this formula yields the correct code word count for any n. As we have no characterization of the individual codewords and the notion of distance between two words is here not well-defined, we have no proof that the codeword set generated above corresponds with the correct code vertex positions. For lower vales of n, we do make the following observations regarding a possible method of defining distance: Suppose A and B are ternary strings. The distance between A and B is 1 if and only if either: **a)** the lowest order bits in A and B are the only ones which differ, **b)** if the lowest two bits in A are different, these two bits are inverted in B or **c)** if the lowest order bits in A are the same, call the first (right to left) different bit in A a and the first different bit in B b. Low n cases indicate that the repeated bit in A is b. D(A,B) = 1 if all a's and b's are complemented in A from a's position to the right. If this is, indeed, the distance measurement for this labelling in two dimensions, the following error correction procedure could be used. Given a string S of length n, first generate the corresponding set of codewords, W_2^n. If $S \in W_2^n$, S is a code word. Generate words S_1 and S_2 at distance one from S by changing the trailing bit of S into the two possibilities. S_3 can be generated by the following algorithm: Let a be the first bit in S differing from the lowest order bit of S b. Complementing all a's and b's through the a position appears to, in all cases observed, generate S_3. One and only one of S, S_1, S_2 and S_3 will be in the set W_2^n. This will be the word S corrects to.

Conclusion

We have shown that the Gray code efficiently labels the one-dimensional graph with easy error-correcting and decoding procedures. We have proven that our codeword structure forms a perfect-one error-correcting code for all $n \geq 0$, and we have also proven that our codeword structure is unique no matter the dimension. Also, we have given a way of labeling the three-dimensional graph which is easily definable and works for n small. However, once we get beyond n=4, the pictures of the tetrahedron become increasingly difficult to draw. Perhaps a program to generate the higher string lengths will be helpful in proving that this labeling does indeed work. These codes do not have any direct applications as with the hypercube representation, however, they are easy to construct and decode, and intuitively represented by nice recursively constructed graphs.

References

[1] P. Cull and I. Nelson. *Error-Correcting Codes on the Towers of Hanoi Graphs.* Technical Report-NSF Grant DMS 93-00-281. Department of Computer Science, Oregon State University. Corvallis, Oregon. August 1995.

[2] R.W. Doran. *The Gray Code.* Technical Report-131. Department of Computer Science, The University of Auckland. Auckland, New Zealand. 1997