Perfect Codes, *NP*-Completeness, and Towers of Hanoi Graphs*

Paul Cull
Department of Computer Science
Oregon State University
Corvallis, Oregon USA
97331–3202
pc@cs.orst.edu

Ingrid Nelson
College of William and Mary
Williamsburg, Virginia
23187

Abstract

The set of codewords for a standard error-correcting code can be viewed as subset of the vertices of a hypercube. Two vertices are adjacent in a hypercube exactly when their Hamming distance is 1. A code is a perfect-error-correcting code if no two codewords are adjacent and every non-codeword is adjacent to exactly one codeword. Since such a code can be described using only vertices and adjacency, the definition applies to general graphs rather than only to hypercubes. How does one decide if a graph can support a perfect 1-error-correcting code? The obvious way to show that such a code exists is to display the code. On the other hand, it seems difficult to show that a graph does not support such a code. We show that this intuition is correct by showing that to determine if a graph has a perfect 1-error-correcting code is an NP-complete problem. The proof is by reduction from 3-SAT. To show that perfect codes in graphs is not vacuous, we give an infinite family of graphs so that each graph in the family has a perfect 1-error-correcting code. Our graphs are based on the Towers of Hanoi puzzle, so that, each vertex is a configuration of the puzzle and two vertices are adjacent when they are one legal move apart. We give a recursive

*Supported in part by NSF Grant DMS 93–00–281

construction which determines which vertices are codewords. There is a natural correspondence between the hypercube vertices and the binary strings, and there is a natural correspondence between Tower of Hanoi configuration and ternary strings. Our recursive construction also specifies which ternary strings are codewords. We characterize the codewords as the set of ternary strings with an even number of 1's and an even number of 2's. As part of this characterization, we show that there is essentially one perfect 1-error-correcting code for each n. There is a unique code when n is even, but the code is only unique up to a permutation of 0, 1, and 2 when n is odd. We show that error-correction can be accomplished by a finite state machine which passes over the ternary string twice, and that this machine is fixed independent of the length of the string. Encoding and decoding are the mappings between integers and codewords, and vice-versa. While algorithms for such mappings can be derived directly from the recursive construction, we show that encoding/decoding can be carried out by multiplication/division by 4 and error-correction. So error-correction, encoding, and decoding can all be done in time $\Theta(n)$ for code strings of length n in these codes.

1 Introduction

Since real machines can malfunction in various ways, people have devised error-correcting codes that will enable them to obtain desired information even when that information has been corrupted by machine faults. A wide variety of codes are used in various areas of communication and computation. The prototypic code of Hamming [Ham50] is special in a number of ways. It is a binary code so the only symbols used are 0 and 1. The distance between words is the Hamming distance, the number of bits at which the two words differ. This implicitly defines the code as a subset of some of the vertices of the hypercube. The code is linear, so that coding can be accomplished by a vector matrix multiplication. The code is, also, perfect in that every word is either a codeword or is adjacent to exactly one codeword. With a great deal of effort, researchers have been able to discover all of the perfect, binary, linear, Hamming distance, codes.[TP71] But in general, other perfect codes may exist. If one gives an explicit listing of codewords for a one-error-correcting code, then it is straight forward and computationally efficient to verify that the listing is really a perfect code. On the other hand, it seems very difficult to show that such a perfect code does not exist. In this paper, we confirm this intuitive idea by defining a code in a graph, and showing that determining if a graph supports a perfect one-error-correcting code is an NP-complete problem.

A graph $G = (V, E)$ is a set of vertices V and set of edges E where each edge is an unordered pair of vertices from V. The distance between two vertices v_i and v_j is the fewest edges which have to be traversed in a path from v_i to v_j. So the distance from a vertex to itself is 0. The distance between two adjacent vertices

is 1. Although other distances are defined, we will not be using them. A code in a graph is a subset $C \subseteq V$. C is called the set of codewords. A code in a graph is error-correcting if for every vertex v there is a unique $C(v) \in C$ so that $C(v)$ is the codeword closest to v. A code in a graph is a perfect one-error-correcting code if for every vertex v, either v is a codeword and is not adjacent to any other codeword, or v is not a codeword and is adjacent to exactly one codeword. Notice that these definitions use only graph properties and do not talk about strings over some alphabet. But as in the Hamming code, we would like to have a natural correspondence between vertices and strings.

In our examples, our strings will be over the ternary alphabet $\{0, 1, 2\}$, rather than the binary alphabet. The distance between strings will be defined by the well-known Towers of Hanoi puzzle. In this puzzle, there are 3 towers which we will call $0, 1$, and 2. There are n disks which are initially stacked on tower 0, with the smallest disk, $disk_1$ on top, and the other disks stacked in order so that the largest disk, $disk_n$, is on the bottom. The puzzle asks one to move the n disks from tower 0 to tower 2 with the restrictions that only one disk is moved at a time and no disk is ever placed on top of a smaller disk. For our purposes, solving this puzzle is not important. We want to use the puzzle to define a graph whose vertices are the configurations of the puzzle and whose edges correspond to single legal moves in the puzzle. To define configurations, notice that each disks is on exactly one tower and that as a result of legal moves, the smaller disks on a tower are always on top of the larger disks on the tower. A configuration of the Towers of Hanoi puzzle with n disks is a vector $d_1, d_2, ..., d_n$ where each $d_i \in \{0, 1, 2\}$. Clearly, there are 3^n configurations. Two configurations are adjacent if one can be reached from the other by a single legal move. Hence adjacent to $d_1, d_2, ..., d_n$ we have $d_1 + 1, d_2, ..., d_n$ and $d_1 + 2, d_2, ..., d_n$ where $+$ is $mod 3$ addition. Further, if d_i is the first component which is not equal to d_1, then adjacent to $d_1, d_2, ..., d_n$ we also have $d_1, ...d_j, ..., d_n$ where $d_j \notin \{d_1, d_i\}$. That is, d_j is the number in $\{0, 1, 2\}$ which is neither d_1 nor d_i. So we have a graph with these 3^n configurations as vertices, and two vertices are adjacent in this graph if they are one legal move apart. Notice that most vertices in this graph have 3 neighbors, but there are exceptional vertices which have only two neighbors. Using configurations as vertices gives us a natural correspndence between vertices and ternary strings. The exceptional vertices are those that have strings in which $d_1 = d_2 = ... = d_n$. So if $n \geq 1$, the exceptional vertices will correspond to the strings $00...0, 11...1, 22...2$, and there will be 3 exceptional vertices. (For more about the Towers of Hanoi puzzle see Cull and Ecklund [CJ85]).

In this paper, we will show how to recursively construct a perfect one-error-correcting code on these Towers of Hanoi graphs. For each value of n, we will construct a perfect code. We will characterize these codes as the set of strings with an even number of 1's and an even number of 2's. We will then show that our construction is essentially the only one possible for these graphs, by showing that

there is a unique perfect one-error-correcting code when n is even, and that when n is odd, there are 3 perfect codes but these are isomorphic by permuting $0, 1$, and 2. We will also show that error-correction for these codes can be accomplished by a small (few state) finite state machines which scans the string twice; once to discover what sort of error was made, and once to correct the error. Finally, we will show that the encoding/decoding problems are easy for these codes. Here, encoding means a mapping from the natural numbers to codestrings, and decoding means a mapping from the codestrings to the naturals. Encoding/decoding uses only multiplication and division by 4, and the error-correcting code procedure. So encoding/decoding takes only $\Theta(n)$ time for n character strings.

2 NP-Completeness

Here we show that deciding if a graph has a perfect 1-error-correcting-code is NP-complete.

Definition 2.1 *P-1-ECC: Given a graph $G = (V, E)$ is there a perfect-1-error-correcting code on G? That is, is there a set of codewords C, so that $C \subseteq V$, and so that:*

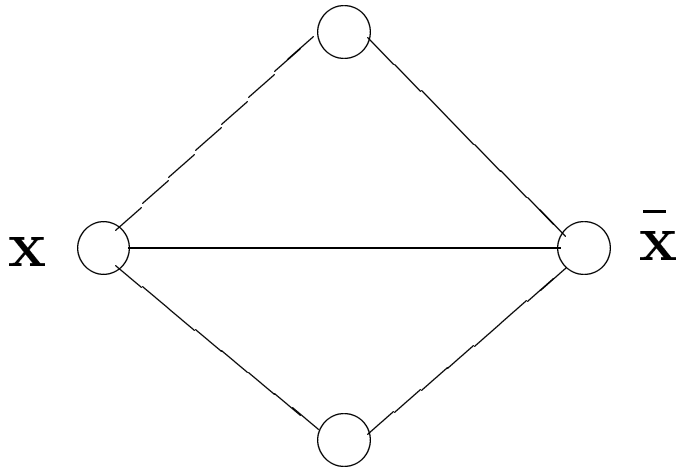
1. $\forall x, y \in C, (x, y) \notin E$ *(no two codewords are adjacent) and*
2. $\forall v \in V - C$ *there is exactly one $x_v \in C$ so that $(v, x_v) \in E$ (every non-codeword is adjacent to exactly one codeword)?*

Theorem 2.1 *P-1-ECC is NP-Complete.*

Proof. : Clearly P-1-ECC is in NP. Given G, guess C and then check that no codewords are adjacent, and that each non-codeword is adjacent to exactly one codeword.

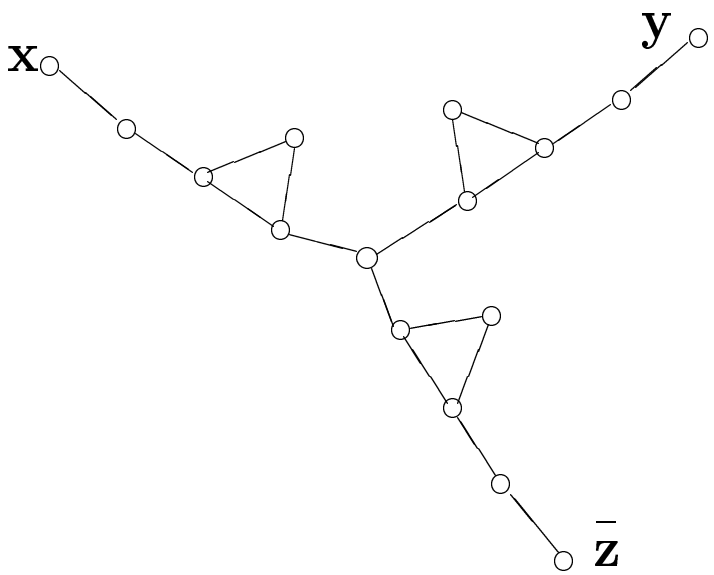
To show that P-1-ECC is complete, we reduce the well-known 3-SAT problem to P-1-ECC. Recall that in 3-SAT we are given an AND of a number of clauses, and each clause is the OR of 3 literals, which are complemented or uncomplemented variables. The question is, is there an assignment of true and false to the variables so that the whole expression evaluates to true? We want to map each instance, S, of 3-SAT to a graph G_S so that G_S has a P-1-ecc iff S has a satisfying assignment. We construct the mapping in two stages. The first stage constructs a "truth setting" subgraph for each variable. For each variable x in S, we construct the

subgraph:

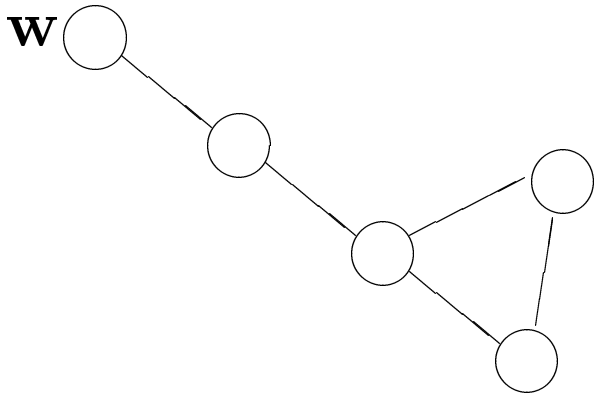


Notice that exactly one of the two vertices labeled x and $\overline{x}$ must be chosen as a codeword. If neither is chosen then each unlabeled vertex will not be adjacent to a codeword, so both unlabeled vertices must be chosen as codewords, but then x and $\overline{x}$ will each be adjacent to two codewords. On the other hand, if both x and $\overline{x}$ are chosen as codewords, both unlabeled vertices will be adjacent to two codewords. Finally, if exactly one of x and $\overline{x}$ is chosen as a codeword, all three remaining vertices will be adjacent to a unique codeword. So finding a P-1-ecc for each of these subgraphs will give a setting for each variable: x is "true" when the vertex x is chosen as a codeword, and x is "false" when $\overline{x}$ is chosen as a codeword.

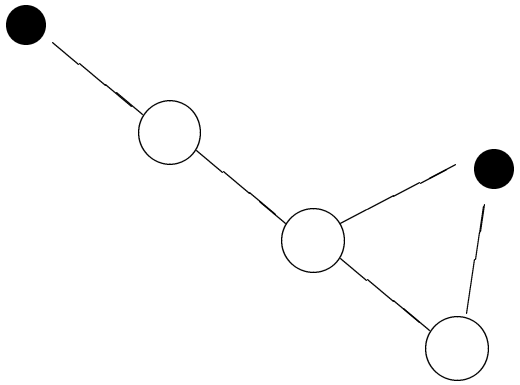
The second stage constructs one "truth checking" subgraph for each clause in S. Assume the clause is $(x \vee y \vee \overline{z})$, we construct the following subgraph in which each of the vertices labeled x or y or $\overline{z}$ is identified with the corresponding labeled vertex in a truth setting subgraph:



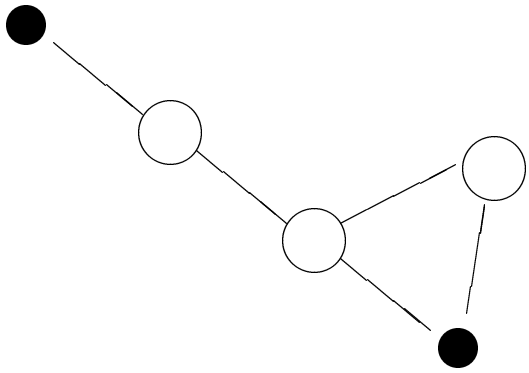
Notice that the subgraph is symmetric in the three labeled vertices so we only have to check that there is a P-1-ecc of the subgraph when at least one of the three labeled vertices is chosen to be a codeword, that is, when one of the three literals in the clause is true. Consider the subgraph



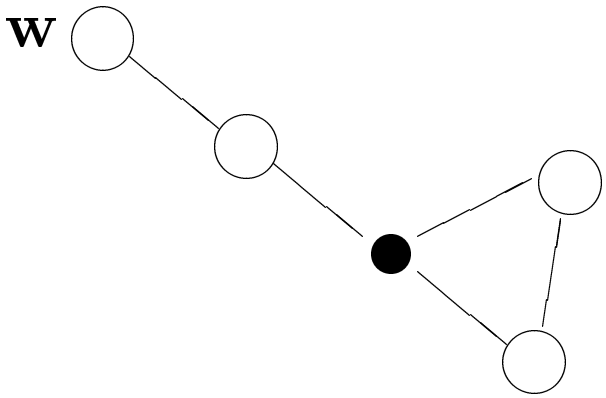
If w is chosen as a codeword, then this subgraph is consistent with a P-1-ecc iff codewords indicated by • are chosen in one of two possible ways, i.e.,



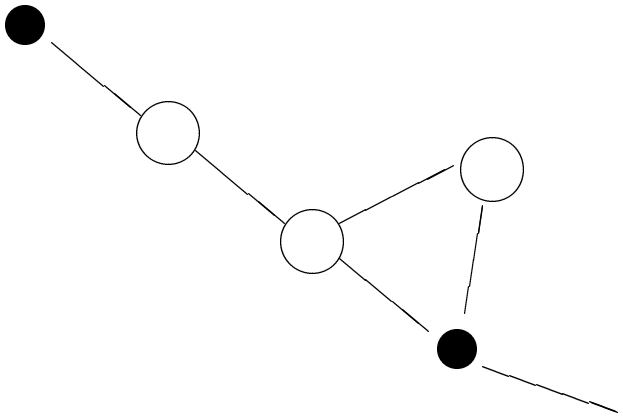
or



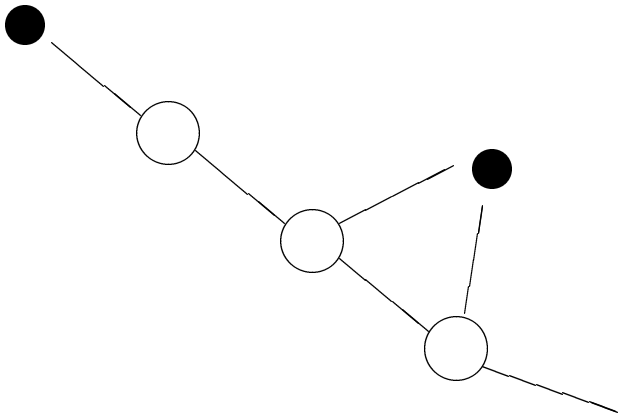
If the literal corresponding to w is chosen to be false, then this subgraph is consistent with a P-1-ecc iff the codewords are chosen in the following way:



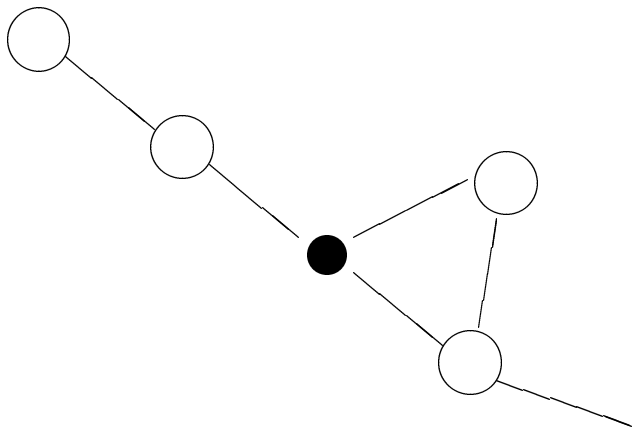
Now for the clause subgraph, if at least one of the literals is true, the subgraph for this literal has codewords chosen as



For the other (if any) true literals the subgraph has codewords chosen as



Any false literal has its subgraph with codewords chosen as



So if there is at least one true literal for the clause, the clause subgraph can have codewords chosen consistent with a P-1-ecc. Notice, in particular that the center vertex is adjacent to exactly one codeword. On the other hand, if all literals are false, the forced choices on the subgraphs leaves the center vertex nonadjacent to any codeword, and the center vertex cannot be chosen as a codeword because then three vertices would be adjacent to the center and also adjacent to another codeword. Hence, the clause subgraph can have codewords chosen consistent with a P-1-ecc iff the clause is satisfied by the assignment of truth values to its literals.

Finally, the whole graph consisting of truth setting subgraphs for each variable, and truth checking subgraphs for each clause, has a P-1-ecc iff the original 3-SAT expression can be satisfied. Notice that the construction of each subgraph is a purely local construction, so the transformation from S to G_S can be computed using only $\log |S|$ space and hence polynomial time. Hence, we have a log space many-one reduction for 3-SAT to P-1-ECC. Of course, this is also a polynomial time many-one reduction. (See Garey and Johnson[GJ79] for further explanation of reductions.) □

3 Constructing the Towers of Hanoi Codes

An error-correcting code on a graph consists of a subset of the vertices (these special vertices are called the 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-correcing code. In the language of graph theory, a perfect one-error-correcting code is a dominating set because every vertex is either a codeword, or shares an edge with a codeword. But a dominating set is not necessarily a perfect one-error-correcting code because two vertices in a dominating set could be adjacent or could both be adjacent to a common vertex.

The number 3 plays a distinguished role in one-error-correcting codes, so a family of graphs based on 3 may be a good place to look for such codes. The Towers of Hanoi puzzle will give such a family of graphs. Recall that in Towers of Hanoi there are 3 towers, which we will call 0, 1, and 2, and n disks which we will call 1, 2, ...,n with 1 for the smallest disk and n for the largest disk. A configuration of the puzzle is specified by an array D where d_i is the tower (0, 1, or 2) which contains the i^{th} disk. According to the rules of the puzzle, only the smallest disk on a tower may be moved, and it can only be placed onto an empty tower or on top of a larger disk. Because of these rules the D array does specify the configuration because on any one tower the disks will be arranged from smallest to largest, and so knowing which disks are on a tower also specifies the order of the disks on the tower.

From the Tower of Hanoi puzzle we can define a graph in which the vertices are the configurations, and in which two vertices are adjacent if the corresponding configurations can be reached from each other by one legal move in the puzzle. For example, 012 represents the configuration in which the smallest disk is on tower 0, the second smallest disk in on tower 1, and the largest disk is on tower 3. The vertex 012 is adjacent to the three vertices 112, 212, and 022 because the first two configurations result from moving the smallest disk from tower 0 to a different tower, and the third configuration results from moving the second smallest disk from tower 1 to tower 2 which is legal because the only disk on tower 2 was the largest disk. In fact, these are the only vertices adjacent to vertex 012 because the only legal moves from this configuration involve moving the smallest disk in two possible ways, and moving the second largest disk in one possible way. More generally, all vertices, except for three special vertices, are adjacent to exactly three other vertices because the smallest disk is on some tower and can be legally moved to either of the other two towers, and, in general, one of the towers which does not contain the smallest disk has a disk which is smaller than all other disks not on the tower with the smallest disk, and so this disk can legally be moved to only one other tower since this disk can not be placed on top of the smallest disk. The

three special vertices correspond to configurations in which all disks are on a single tower. In this special situation, the only legal moves are moving the smallest disk to another tower. So the special vertices are adjacent to exactly two other vertices.

Of course, our description of the T of H graph defines a whole family of graphs, one for each value of n, the number of disks. Pleasantly, there is a pictorial description of the contruction of a T of H graph from smaller T of H graphs. For example, let H_n be the T of H graph for n disks, then:

$$H_n = \quad \begin{array}{ccc} & H_{n-1} & \\ / & & \backslash \\ H_{n-1} & — & H_{n-1} \end{array}$$

That is, the T of H graph for n disks is constructed out of 3 copies of the T of H graph for $n-1$ disks by adding 3 edges which connect the corner vertices in the indicated fashion. These pictures make it easy to see that there are 3^n vertices in H_n, and that the maximum distance between any two vertices is $2^n - 1$.

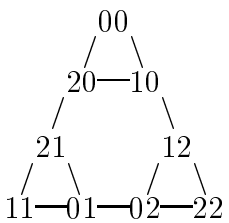
Although the above diagram captures the topology of the graphs, it does not display the labeling which we will need in constructing the codes. Let L_n be the labeled graphs, then:

$$L_{n+1} = \quad \begin{array}{ccc} & RL_n0 & \\ / & & \backslash \\ \uparrow RL_n1 & — & \downarrow RL_n2 \end{array}$$

By this we mean that the labeled graph for $n+1$ disks can be constructed from 3 copies of the labeled graph for n disks. By RL_n we mean the labeled graph which is the mirror image of L_n. In L_n, the lower right vertex is labeled 22....2, and the lower left vertex is labeled 11....1. In RL_n, the lower right vertex is labeled 11....1, and the lower left vertex is labeled 22....2. The top copy, RL_n0, looks like RL_n, but each vertex has a 0 appended to its label. Similarily, $\uparrow RL_n1$ is a copy of RL_n which has been rotated 120 degrees clockwise and has a 1 appended to each label, and $\downarrow RL_n2$ is a copy of RL_n which has been rotated 120 degrees counterclockwise and has a 2 appended to each label. For example,

$n = 1$

$n = 2$



From these labeled graphs, it is relatively straightforward to construct the codes. Since the graphs are constructed in levels, the codes can also be constructed in levels. Let the top level, which consists only of the vertex with label 00....0, be the 0^{th} level. One can start by making this label, i.e. 00....0, a codeword, then, since every codeword must be a distance 3 from every other codeword, codewords will only appear in every third level. Within one of these levels we want the maximum number of codewords which are at distance 3 from each other This process of constructing codewords can be explained with some diagrams. Let G_n be the graph with the same topology as L_n, but with no labels. Instead G_n has a circle around each vertex which corresponds to a codeword. So by placing G_n on top of L_n, one can read off the codewords as the labels of the circled vertices. To show how to construct G_n, we will need another sequence of graphs, which we will call U_n. U_n, like G_n, will have the same topology as L_n, and will have circles around vertices which represent codewords, but U_n is only used in constructing G_n, and does not represent an error-correcting-code. The following diagrams indicate the construction:

$$G_n = \quad \begin{array}{c} G_{n-1} \\ / \quad \backslash \\ \downarrow G_{n-1} — \uparrow G_{n-1} \end{array} \qquad n \text{ even}$$

$$G_n = \quad \begin{array}{c} G_{n-1} \\ / \quad \backslash \\ U_{n-1} — U_{n-1} \end{array} \qquad n \text{ odd}$$

$$U_n = \quad \begin{array}{c} U_{n-1} \\ / \quad \backslash \\ G_{n-1} — G_{n-1} \end{array} \qquad n \text{ even}$$

$$U_n = \quad \begin{array}{c} U_{n-1} \\ / \quad \backslash \\ \downarrow U_{n-1} — \uparrow U_{n-1} \end{array} \qquad n \text{ odd}$$

G_0 =O (a codeword), and U_0 = X (not a codeword).

As in the construction of L_n, the arrows in the construction indicate that the copies of U_{n-1} and G_{n-1} have to be rotated.

We are now in a position to state and prove the main theorem:

Theorem 3.1 *For every $n \geq 0$, G_n defines a perfect one-error-correcting code.*

It is easy to follow the construction and show that G_0, G_1, and G_2 do define perfect one-error-correcting codes. For larger values of n, an inductive proof is needed and it will follow from the slightly complicated lemma which follows.

Lemma 3.2 *For each $n \geq 2$,*
in G_n
a) each uncircled vertex is at distance one from exactly one circled vertex, no circled

vertices are adjacent,
in U_n
b) each noncorner vertex is at distance one from exactly one circled vertex, no circled vertices are adjacent,
c) if n *is* odd*, all three corner vertices are at distance* 2 *from exactly one circled vertex.*
d) if n *is* even*, the apex vertex is at distance* 2 *from exactly one circled vertex, and the other two corner vertices are at distance* 1 *from exactly one circled vertex.*

Proof. : Checking these properties for $n = 2$ is easy. For larger values of n, we will work through the four cases of the construction and show that these properties holding for $n-1$ imply that these properties hold for n.

For G_n with n *even*, each vertex is in a G_{n-1} for which (a) holds. The only possible problem is that some corner vertices of G_{n-1} are also attached to a vertex in another G_{n-1}, but since none of these vertices are circled, each such vertex is at distance 1 from only one circled vertex. So (a) holds for G_n. For G_n with n *odd*, (a), (b), and (d) imply that each uncircled vertex is at distance 1 from exactly one circled vertex. In particular, (d) implies that the apex vertices of the U_{n-1}'s are at distance 2 from a circled vertex within their U_{n-1}, and so these apex vertices are at distance 1 from only the circled corner vertices of G_{n-1}. So (a) holds for G_n.

For U_n with n *even*, the previous argument shows that (b) holds. Condition (c) holds vacuously. Condition (d) follows from (c) and (a).

For U_n with n *odd*, (b) implies (b) except for the corner vertices of the U_{n-1}'s. Condition (d) implies (b) for the corner vertices of the U_{n-1}'s which are attached to another U_{n-1}. Condition (d) also implies that the corner vertices of U_n which were apex vertices of U_{n-1} are at distance 2 from exactly one circled vertex. So condition (c) holds. Condition (d) holds vacuously. □

The theorem now follows because each vertex is either a codeword vertex, or at distance 1 from exactly one codeword vertex, and no two codeword vertices are adjacent.

4 Characterizing the Codewords

The previous section gave a graphical construction of the codes. In this section, we give a characterization of the codewords as ternary strings. To do so, we use the three functions $\#_0$, $\#_1$, and $\#_2$ which count the number of occurrences of 0, 1, and 2 as characters in a string. So, for example, $\#_0(0210) = 2$, $\#_1(0210) = 1$, and $\#_2(0210) = 1$. The T of H codewords as strings can be generated by a recursive procedure which follows directly from the graphs in section 3. Here let G_n mean the set of codewords as strings of length n, and let U_n be the auxillary set of strings

of length n, then for even n

$$G_n = G_{n-1} \circ 0 \cup T(G_{n-1}) \circ 1 \cup T^2(G_{n-1}) \circ 2$$

$$U_n = U_{n-1} \circ 0 \cup T(U_{n-1}) \circ 1 \cup T^2(U_{n-1}) \circ 2$$

and for odd n

$$G_n = G_{n-1} \circ 0 \cup , _1(U_{n-1}) \circ 1 \cup , _2(U_{n-1}) \circ 2$$

$$U_n = U_{n-1} \circ 0 \cup , _1(G_{n-1}) \circ 1 \cup , _2(G_{n-1}) \circ 2$$

where $G_{n-1} \circ 0$ means add a 0 to the right end of each string in G_{n-1}; where $\cup$ means set union; T is an operator which replaces each character in a string by another character following the permutation $0 \to 1 \to 2 \to 0$; T^2 means apply T twice, that is, replace characters by the permuation $0 \to 2 \to 1 \to 0$; $, _1$ changes characters by $2 \rightleftharpoons 0$; and $, _2$ changes characters by $1 \rightleftharpoons 0$. Of course, when an operation is applied to a set, the operation is applied to each string in the set. The codewords are characterized by the following theorem:

Theorem 4.1 *The set of codewords G_n of length n is the set of ternary strings of length n which satisfy $\#_1 \equiv \#_2 \equiv 0(mod 2)$. The auxilliary set U_n is the set of ternary strings of length n which satisfy $\#_1 \equiv \#_2 \equiv 1(mod 2)$.*

Proof. The proof is a straightforward induction. Starting with $G_0 = \{\varepsilon\}$, the set which contains the null string, and $U_0 = \phi$, the empty set. It is easy to verify that $G_1 = \{0\}$, $U_1 = \phi$, $G_2 = \{00, 11, 22\}$, and $U_2 = \{21, 12\}$, and see that these sets satisfy the hypotheses. The rest of the proof simply plugs in the hypotheses and mechanically checks them. For instance, if n is odd, the hypothesis asserts that $\#_1 \equiv 0$ for all strings in G_n, and this requires

$$\#_1(G_{n-1}) \equiv \#_1(, _1(U_{n-1})) + 1 \equiv \#_1(, _2(U_{n-1})) \equiv 0.$$

But $\#_1(G_{n-1}) \equiv 0$ by hypothesis, and

$$\#_1(, _1(U_{n-1})) + 1 \equiv \#_1((U_{n-1})) + 1 \equiv 1 + 1 \equiv 0.$$

and

$$\#_1(, _2(U_{n-1})) \equiv \#_0((U_{n-1})) \equiv n - 1 - 1 - 1 \equiv n - 1 \equiv 0$$

because n is odd. Of course, one also needs to check that all strings which satisfy $\#_1 \equiv \#_2 \equiv 0(mod 2)$ are in G_n. But any such string which ends in a 0 has its first $n-1$ characters satisfying the hypothesis that requires the $n-1$ character string to be in G_{n-1}. Similarly, any such string which ends in a 1, must have an odd number of $1's$ and an even number of $2's$ in its first $n-1$ characters, and since $n-1$ is even, an odd number of $0's$. Now changing $0's$ to $2's$ and vice-versa, yields a string of length $n-1$ with an odd number of $1's$ and an odd number of $2's$ and by hypothesis such a string is in U_{n-1}. Similarly, one can check to see that any string with $\#_1 \equiv \#_2 \equiv 0(mod 2)$ and ending in a 2 must be derived from a string in U_{n-1} in the manner specified. □

5 Counting the Codewords

Since the T of H codes are perfect one-error-correcting codes, about 1/4 of the strings of length n will be codewords. The exact count is given in the following theorem.

Theorem 5.1 *The number of codewords in the T of H code is:*

$$\frac{3^n + 2 + (-1)^n}{4} = \begin{cases} (3^n+3)/4 & n \text{ even} \\ (3^n+1)/4 & n \text{ odd} \end{cases}$$

This result can be established in two different ways. One way is to derive a difference equation directly from the construction. The other way is to derive a generating function from the characterization. For the first method, let g_n be the number of words in G_n and let u_n be the number of words in U_n. Then from the generating construction we have:

$$\begin{aligned} g_n &= 3g_{n-1} && n \text{ even} \\ g_n &= g_{n-1} + 2u_{n-1} && n \text{ odd} \\ u_n &= 2g_{n-1} + u_{n-1} && n \text{ even} \\ u_n &= 3u_{n-1} && n \text{ odd} \end{aligned}$$

with $g_0 = 1$ and $u_0 = 0$. Noticing that $g_n - u_n = g_{n-1} - u_{n-1} = 1$ for all n, gives $g_n = 3g_{n-1} - 1 + (-1)^n$ which has the solution given in the theorem.

For the second method, the trinomial $(x+y+z)^n$ will serve as a generating function for all string over $\{0,1,2\}$. When we interpret say x as 0, y as 1, and z as 2, the coefficient of $x^i y^j z^r$ will count the number of strings with i $0's$, j $1's$ and r $2's$. Let us define S_{00} as the number of strings in which the number of $1's$ is even and the number of $2's$ is even. Similarly, we define S_{01}, S_{10}, S_{11} to be the number of strings in which the parity of the number of $1's$ is the first subscript and the parity of the number of $2's$ is the second subscript. Then the generating function gives:

$$\begin{aligned} (1+1+1)^n &= S_{00} + S_{01} + S_{10} + S_{11} &&= 3^n \\ (1-1+1)^n &= S_{00} + S_{01} - S_{10} - S_{11} &&= 1^n \\ (1+1-1)^n &= S_{00} - S_{01} + S_{10} - S_{11} &&= 1^n \\ (1-1-1)^n &= S_{00} - S_{01} - S_{10} + S_{11} &&= (-1)^n \end{aligned}$$

Solving these equations for S_{00} gives the results reported in the theorem.

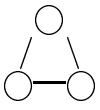
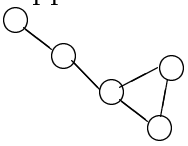
This count of the codewords also gives us a proof that the T of H codes are nonlinear. Recall that a code is linear if the set of codeworda is a subspace of the vector space V^n which is the set of n component vectors over the set of scalars V.

More simply, if V^n with an appropriate addition is a group, then a code C is a linear code iff C is a subgroup of V^n.

It is well known that the size (number of elements) of a subgroup must divide the size of the group. For the T of H codes $V^n = \{1, 2, 3\}^n$, and so $|V^n| = 3^n$. As we have just shown $|C| = (3^n + 3)/4$ or $|C| = (3^n + 1)/4$. In either case, if $n > 2$, $|C|$ does not divide $|V^n|$, so C cannot be a subgroup, and hence C is a nonlinear code. In the special cases, $n = 1$ or $n = 2$, $|C|$ does divide $|V^n|$ and in these special cases C is a linear code. We summarize these results in the following theorem.

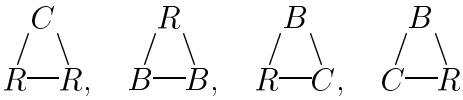
Theorem 5.2 *If $n > 2$, the T of H codes are nonlinear.*

6 Uniqueness

Even though we have established that each T of H graph has a P-1-ecc, as we will abbreviate perfect one-error-correcting code, it is plausible that a graph might support several distinct P-1-eccs. For example  has 3 different P-1-eccs, and  has 2 different P-1-eccs even if we specify that the top vertex must be chosen as a codeword. Thus we would like to prove that the T of H graph has only one P-1-ecc. Unfortunately, this claim is false. As the above triangle example shows there are T of H graphs with several different P-1-eccs. Instead, we will show the T of H graph has only one P-1-ecc in which the top vertex is chosen as a codeword.

A *consistent 3-labeling* is an assignment of a label from {C, B, R} to each vertex so that the following restrictions are met:

a) each triangle of the graph is labeled in one of the following four ways:



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

c) no two vertices labeled C are adjacent

d) vertices labeled B in the bottom corners are not adjacent to a C, but every other B is adjacent to a C

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

Lemma 6.1 *: A P-1-ecc of H_n induces a consistent 3-labeling of H_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. Consider the top vertex. Either it is a codeword, and hence is labeled C, or by the P-1-ecc assumption, it is the predecessor of a codeword, and hence is labeled B. Now one can label any unlabeled vertices by proceeding down the T of H graph labeling the succcessors of R's with B, and labeling the successor of B's with R. Since each vertex (except the top vertex) has exactly one predecessor, after this process all vertices will be labeled. Now we have to check that all the restrictions are satisfied. For the restrictions in (a), consider the top vertex of each triangle. If the top vertex is labeled C then the other vertices are successors of a codeword, and hence are labeled R, giving $\begin{array}{c} C \\ / \quad \backslash \\ R\text{—}R \end{array}$. If the top vertex is labeled R, then neither of its successors is a C because then the top vertex would be labeled B instead of R. Further, neither successor is labeled R because neither is a successor of a C or of a B. This leaves $\begin{array}{c} R \\ / \quad \backslash \\ B\text{—}B \end{array}$ as the only possibility. Finally, if the top vertex is a B, its predecessor cannot be a C and from the P-1-ecc condition, the B vertex must be adjacent to exactly one codeword so one of its successors is a C, and the other successor is an R. This allows the two possibilities $\begin{array}{c} B \\ / \quad \backslash \\ R\text{—}C \end{array}$ and $\begin{array}{c} B \\ / \quad \backslash \\ C\text{—}R \end{array}$. The restrictions in (b) follow directly from the construction of the labeling. The restricitons in (c) follow because in a P-1-ecc no two codewords are adjacent. Restrictions (d) and (e) follow because from the P-1-ecc each non-codeword is adjacent to exactly one codeword and so each B and R is adjacent to exactly one C. The exceptions in (d) and (e) cannot arise. As shown above, the top vertex cannot be labeled R. A bottom corner cannot be labeled B because from (a) it would be in a $\begin{array}{c} R \\ / \quad \backslash \\ B\text{—}B \end{array}$ triangle and hence would not be adjacent to a codeword as required by the P-1-ecc. □

Next we show that a consistent 3-labeling can only be built from smaller consistent 3-labelings.

Lemma 6.2 *: Any consistent 3-labeling of H_n gives a consistent 3-labeling for each of H_n's constituent H_{n-1}'s.*

Proof. : Since

$$H_n = \begin{array}{c} H_{n-1} \\ / \qquad \backslash \\ H_{n-1}\text{—}H_{n-1}, \end{array}$$

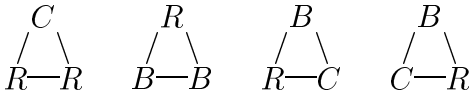
the restrictions (a), (b), and (c) for H_n imply the same restrictions on each of the three H_{n-1}'s. For (d), if any of the three H_{n-1}'s has a B in a bottom corner, then

by (a) this B is part of an triangle. and hence is not adjacent to a C in the H_{n-1}. All other B's are not corner vertices of either H_n or H_{n-1} and by (d) for H_n, these B's are adjacent to a C. For (e), if any R is a top vertex of an H_{n-1} then it is either the top vertex of H_n and so by (e) for H_n is not adjacent to any C, or it is internal for H_n and hence adjacent to exactly one C, but by (a) this R

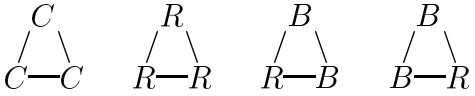
is the apex of a triangle and thus in the H_{n-1} it is not adjacent to a C. All other R's are adjacent to exactly one C by (e) for H_n. □

Next we show that only a few consistent 3-labelings of H_n exist.

Lemma 6.3 *: For each $n \geq 1$, there are exactly four possible consistent 3-labelings of H_n and they have the following forms: for* odd *n, the corner vertices have one of the following four patterns:*

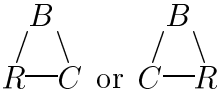


for even *n, the corner vertices have one of the following four patterns*

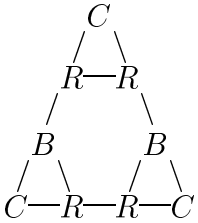


Proof. : For $n = 1$, H_n is a triangle and so by (a) of the definition the only consistent 3-labelings are exactly the four listed. Of course, (b), (c), (d), and (e) are satisfied by these labeled triangles. For $n > 1$, the previous lemma says that consistent 3-labelings can only be built from smaller consistent 3-labelings. In particular, for even n, the consistent 3-labelings can only be built from the

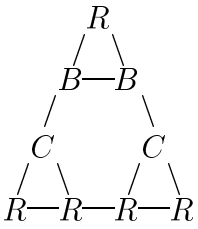
consistent 3-labelings for odd n. Starting with the pattern for the top H_{n-1} in H_n, the successor restrictions require the lower H_{n-1}'s to be labeled with



patterns or . Of the four ways these labelings could be attached together one is ruled out by (c) no two adjacent C's, and two are ruled out by (e) an R is adjacent to exactly one C. The remaining possibility is

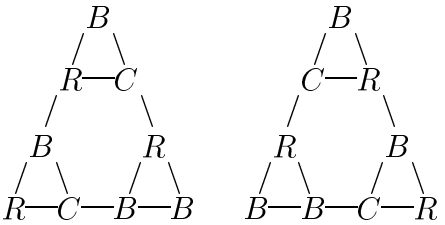


and this labeling satisfies (a), (b), (c), (d), and (e). This is a consistent 3-labeling for H_n with the pattern  which is one of the listed possibilities for even n. Next if the top vertex is R, the top H_{n-1} has the pattern and by (d) the bottom B's are not adjacent to a C. To satisfy (d) for H_n, the lower H_{n-1}'s must have the pattern  So the only possibility is



and this inherits the properties (a), (b), (c), (d), and (e) from those properties for H_{n-1}. Thus, we have one consistent 3-labeling of the pattern  for H_n when n is even, and this is one of the listed possibilities.

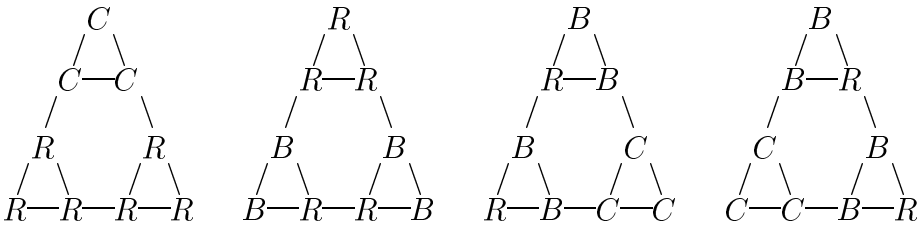
Similarly, if the top vertex of H_n with n even is labeled B, then the possible patterns for the top H_{n-1} are  and  and so the possible patterns for the bottom H_{n-1}'s are  ,  , and  . But by (d), the B from  that gets connected, must be connected to a C. Hence the possibilities are

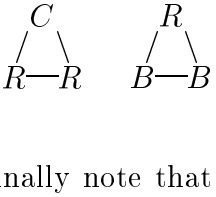


These labelings will inherit (a), (b), (c), (d), (e) from those properties for the H_{n-1}'s. So we have two more consistent 3-labelings for H_n, and they have the patterns  and  which are listed possibilities.

Now for n odd and $n > 1$, we must start with one of the four even patterns for H_{n-1}, and build the possible labelings for H_n. Using the successor rules, the fact

that if a B in a bottom corner is connected it must be connected to a C, and an R cannot be connected to a C, we find only the following four possibilities:



It is easy to check that each of these inherits properties (a), (b), (c), (d), (e) from those properties for H_{n-1}'s. These labelings have the forms  which are exactly the forms listed for n odd. Finally note that there is exactly one consistent 3-labeling for each of these patterns. And so the lemma is established. □

Finally we are ready for the uniqueness theorem for Towers of Hanoi codes.

Theorem 6.4 *For each $n \geq 0$, there is a unique (up to rotation by 120°) perfect-1-error-correcting code for the Towers of Hanoi graph H_n. If the top vertex is required to be a codeword, then the code is strictly unique.*

Proof. : For $n = 0$, H_n is a single vertex, and the code in which this vertex is a codeword is a unique P-1-ecc. For $n \geq 1$, by lemma 6.1 if there is such a code then there must be a consistent 3-labeling. By lemma 6.3 there are four such labelings for each n. For even n, three of these labelings cannot give P-1-ecc's because they 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 labeled C a codeword, and each vertex labeled R or B a non-codeword. Hence for even n, the only pattern which gives a P-1-ecc is  and it is easy to check that the code is rotationally symmetric. For odd n, one of the four labelings contains a bottom B and so cannot give a P-1-ecc. There is only one labeling which has a C at the top and so is unique. The other two lablelings will simply give rotations of this code. □

7 Error Correction

A reasonable code should have a reasonable error correction procedure. That is, given a string it should be easy to calculate the closest codeword to that string.

For the T of H code, detecting an error is certainly easy, because there is no error exactly when $\#_1 \equiv \#_2 \equiv 0(mod 2)$. So if at least one of these equalities is violated there is an error. It is only slightly more difficult to find where the error occurred and correct it.

Since we are dealing with only a single error, and this corresponds to a single move in the T of H puzzle, we should consider what moves can occur. The smallest disk, disk 1, is on some tower, and the puzzle rules allow this disk to be moved to either of the other two towers. For example, if disk 1 is on tower 2, and $\#_1 \equiv 1 \equiv \#_2(mod 2)$, then moving disk 1 to tower 1 will correct $\#_1$ and $\#_2$ and the resulting string will be a codeword. More specifically, assume that the given word is

$$D = d_1 d_2 ... d_n = 2 d_2 ... d_n$$

and $\#_1(D) \equiv 1 \equiv \#_2(D)$, then the corrected word $\hat{D}$ is $1 d_2 ... d_n$ and $\#_1(\hat{D}) \equiv 0 \equiv \#_2(\hat{D})$.

If all the disks were on a single tower, then the above would be the only possibility, but if there are disks on two or three towers, then another kind of move is possible. Consider the two towers which do not contain disk 1. The smallest disk on these towers can be legally moved from its present tower to the other tower. We have just shown that only three or two moves are possible from each configuration, and these moves involve either disk 1, or the smallest disk on the towers not containing disk 1.

So the error correcting procedure will determine which two towers are involved in the error by calculating $\#_1$ and $\#_2$. Then if disk 1 is on one of these two towers, the error is corrected by moving disk 1 to the other tower. If disk 1 is not on either of these towers, then the smallest disk on these two towers will be moved from one tower to the other.

This discussion in terms of disks and towers is somewhat metaphoric since we want to process strings. Clearly, calculating $\#_1$ and $\#_2$ can be done by scanning a string. If the error-correction should move the smallest disk, then the value of the first character in the string is changed. If the error-correction should move a disk other than the smallest disk, then the string is scanned until a character that is different from the first character is encountered, and then this different character is changed. In either case, two towers are involved and only the single position in the string which represents the smallest disk on these two towers is changed.

The decoding can proceed via the following algorithm. The algorithm will compute $\#_0$, $\#_1$, and $\#_2$ but only $\#_1$ and $\#_2$ are needed. The variables f_0, f_1, and f_2 serve to indicate the relative ordering of appearances of 0, 1, and 2 in the string. Thus $f_j = 1$ will indicate that $d_1 = j$, while $f_k = 2$ will indicate that k is the value of the first character in the string which differs from d_1, that is, $d_1 = d_2 = ... = d_{l-1} \neq d_l$ and $d_l = k$.

INPUT: STRING $D = d_1 d_2 ... d_n$

$f_0 = f_1 = f_2 = 0 \quad \#_0 = \#_1 = \#_2 = 0$
FOR $i = 1$ TO n
$\#_{d_i} = \#_{d_i} + 1(mod 2)$
IF $f_{d_i} = 0$
THEN $f_{d_i} = 1 + max(f_0, f_1, f_2)$
ENDFOR
IF $\#_1 = 1$ and $\#_2 = 1$
THEN IF $f_1 < f_2$
THEN scan until first 1 is found and change it to 2
ELSE scan until first 2 is found and change it to 1
IF $\#_1 = 1$ and $\#_2 = 0$
THEN IF $0 < f_0 < f_1$
THEN scan until first 0 is found and change it to 1
ELSE scan until first 1 is found and change it to 0
IF $\#_1 = 0$ and $\#_2 = 1$
THEN IF $0 < f_0 < f_2$
THEN scan until first 0 is found and change it to 2
ELSE scan until first 2 is found and change it to 0

Note that $\#_i$ is only computed $mod 2$, and f_j must be in {0,1,2,3}, so a machine with a very limited finite memory is sufficient to do the correction. We summarize this result in the following theorem.

Theorem 7.1 *The Towers of Hanoi code can be decoded by a finite memory machine which scans the input string twice.*

8 Coding and Decoding

What we have done in the previous section is sometimes called decoding, but we prefer to call it error-correction, and reserve the terms coding and decoding for the mapping between the integers and the codestrings. That is, we want two mappings CODE and DECODE so that:

$$CODE : \{0, 1, ..., |G_n| - 1\} \to G_n$$

$$DECODE : G_n \to \{0, 1, ..., |G_n| - 1\}$$

An immediate solution to this problem is available from the recursive description of the codewords. For example, to code when n is even, we can use the formula

$$G_n = G_{n-1} \circ 0 \cup T(G_{n-1}) \circ 1 \cup T^2(G_{n-1}) \circ 2$$

Then if one is given the integer I, one can determine which third I belongs to and thus determine one trit. One could then recursively find the other $n - 1$ trits. Of

course, one would also use the other formulas when n is odd, or when one has to generate a string in the auxillary set U_n. This technique is not entirely satisfying because it may take a long time. Since n-trit numbers are being compared, the comparisons could take $\Theta(n)$ time. Further, the string transformations T and , may take $\Theta(n)$ time. Since this method would use $\Theta(n)$ time per step and there are n steps to generate a string, a total of $\Theta(n^2)$ time could be used to produce a code string. Some standard tricks can be used to speed up this process. Instead of applying the string transformation to $n-1$ bits in each step, one could generate a total transformation as well as the characters of the string. Then when one character is being found, the total transformation is applied to only one character, and a new total transformation is calculated by composing the transformation specified by the generation rule with the total transformation calculated so far. Luckily, this composition is easy, because the transformations all come from the set of permutations on 3 elements, and hence the total transformation has only one of 6 possible values. For comparing strings, there is also a simple trick. One simply notices that size of the G_n and U_n sets can be expressed in ternary using only a few non-zero trits. So these tricks can result in $\Theta(n)$ average case time to generate a string from an integer, and it is possible that one could also show that these tricks could result in $\Theta(n)$ worst case time.

The above seems like a lot of effort, and the single characterization of the codewords suggests that there should be a simple way to $CODE$ and $DECODE$. The clue is that to obtain the set sizes with a few trits, one needs to multiply the set size by 4. So now, the question is: What happens when one multiplies an integer in base 3 by 4? In many cases, one obtains a codeword. But this is not always the case. Since the code is perfect one-error-correcting, even when multiplying by 4 does not yield a codeword, the product number represents a node which is adjacent to a codeword. Hence, $ERROR-CORRECT(4*I)$ will always produce a codeword. The codeword so produced will be different for different I's, if every distance one neighborhood of a codeword contains exactly one number which is divisible by 4. And if this condition holds, then one can decode a codeword x by $I = N(x)/4$ where $N(x)$ is the unique element in the neighborhood of x which is divisible by 4. These considerations lead to the following theorem:

Theorem 8.1 *$CODE$, a one-to-one onto map from the integers in $\{0, 1, ..., |G_n| - 1\}$ to the set of codewords G_n is given by $CODE(I) = ERROR - CORRECT(4 * I)$. $DECODE$, a one-to-one onto map from the set of codewords, G_n, to the integers in $\{0, 1, ..., |G_n| - 1\}$ is given by $DECODE(x) = N(x)/4$ where $N(x)$ is the unique number divisible by 4 which is associated with a node in the neighborhood of x.*

Proof. To prove this theorem, we only have to show that the neighborhood of a codeword contains exactly one node whose associated string when considered as a base 3 number is divisible by 4. A string consists of the ternary digits (trits) $d_0, d_1, ..., d_{n-1}$ and has the associated number $\sum_{i=0}^{n-1} d_i 3^i$. (We could interpret the

trits left-to-right or right-to-left, but we will find left-to-right convenient below.). Since $3 \equiv -1(mod4)$, each number $mod4$ can be written as

$$\sum_{i=0}^{n-1} d_i(-1)^i = 0 \cdot \sum_{i|d_i=0} (-1)^i + 1 \cdot \sum_{i|d_i=1} (-1)^i + 2 \cdot \sum_{i|d_i=2} (-1)^i$$

By the characterization theorem, each codeword has an even number of 1's and even number of 2's. So we would like to know what $\sum_i (-1)^i$ is, when there are an even number of i's in the sum. Since the number of odd i's and the number of even i's are equivalent $mod2$, this summation can only give $EVEN - EVEN$ or $ODD - ODD$, which will always produce an even number or, equivalently, an number which is 0 or 2 $mod4$. So if we let D be the $mod4$ number associated with a codeword, then $D \in \{0, 2\}$. By the rules of the T of H puzzle, each neighborhood will consist of 3 or 4 nodes; the node itself, two nodes that correspond to moving the smallest disk, and possibly a node which corresponds to moving the smallest disk which is not on the same tower as the smallest disk. The numbers formed by moving the smallest disk will simply change the low order trit. The other move will change one trit d_i to the value that is neither d_0 nor d_i. In the three cases based on the value of the low order trit we have as the numbers in the neighborhood:

$$\begin{array}{lllll} \mathrm{d}_0 = 0: & D, & D+1, & D+2, & D \pm 1 \\ \mathrm{d}_0 = 1: & D, & D-1, & D+1, & D \pm 1 \\ \mathrm{d}_0 = 2: & D, & D-1, & D-2, & D \pm 1 \end{array}$$

We have placed a $\pm$ in the fourth number because whether to add or subtract depends on the parity of i. In each case, the first, second, and third numbers are in the neighborhood, and the fourth number will be in the neighborhood when the codeword is not a corner vertex. In each case, exactly one of D and $D \pm 2$ will be divisible by 4, and the other numbers are not divisible by 4. In the special case when $d_0 = 1$ and $D \pm 2$ is not in the neighborhood, $d_0 = d_1 = ...d_{n-1} = 1$ and n is even, so $D = \sum_{i=0}^{n-1}(-1)^i = 0$. Hence each neighborhood of a codeword contains exactly one number divisible by 4. □

As one would expect CODE and DECODE behave as inverses because

$$\begin{aligned} CODE(DECODE(x)) &= CODE(N(x)/4) \\ &= Error - Correct(4 \cdot N(x)/4) \\ &= Error - Correct(N(x)) = x \end{aligned}$$

and

$$\begin{aligned} DECODE(CODE(I)) &= DECODE(Error - Correct(4 \cdot I)) \\ &= N(Error - Correct(4 \cdot I))/4 \\ &= 4 \cdot I/4 = I \end{aligned}$$

To be useful $CODE$ and $DECODE$ should be able to be quickly computed. In fact, both $CODE$ and $DECODE$ can be computed in $\Theta(n)$ when applied to numbers with n trits or strings with n ternary characters. This run time follows because multiplication and division by 4, error correction, and finding neighbors can all be carried out in $\Theta(n)$ time. Error correction was shown to be fast in the last section where it was shown that a two pass finite state machine can do error correction. Finding neighbors is fast because two of the neighbor are found by simply changing the low order trit, and the other neighbor can be found by scanning the string for the first d_i which is not d_0 and changing d_i. Of course, multiplication and division by 4 are rapid operations.

9 Conclusion

Some of the topics mentioned in this paper have a long history. For example, the idea of perfect codes in graphs goes back at least to Biggs [Big73]. The search for perfect codes has a long and fascinating history which can be found in the standard coding theory references like Hill[Hil86] and MacWillaims and Sloane [MS77].

NP-completeness is often used to explain why a problem is difficult. The standard reference on NP-completeness is Garey and Johnson [GJ79]. Some more examples of NP-complete problems about codes can be found in [BMvT78] [Bar94] [NH81] [BN90] and [Bar98].

Since the proof of NP-completeness for perfect codes in graphs is so straightforward, it would be surprising if it had not been discovered before. Our literature search found mention of the result in [Kra87], [KK88], [Kra94], and [Kra91]. The question of what restrictions can be placed on a family of graphs while still leaving the existence of perfect codes on those graphs as an NP-complete problem was investigated in [Kra91]. But we were unable to find a publication which actually gave the NP-completeness proof. We include the proof in this paper with the hope that, because of its simplicity, it will become the standard example of a computationally hard problem about codes.

To show that perfect codes in graphs is not a vacuous idea, we have presented an infinite family of graphs based on the Tower of Hanoi puzzle, and proved that these graphs all have perfect one-error-correcting codes. Further, we characterized the codewords as those ternary strings with even numbers of 1's and even numbers of 2's. We also counted the codewords, and proved that these codes are essentially unique. We showed that error correction can be accomplished by two passes of a small finite state machine. We showed that coding and decoding strings of length n can be done in $\Theta(n)$ time. We also note that these codes are nonlinear, but that they are computationally simpler than all but the simplest linear codes.

References

[Bar94] A. Barg. Some new NP-complete coding problems. *Problems of Information Transmission*, 30(3):44–49, 1994.

[Bar98] A. Barg. Complexity issues in coding theory. *Handbook of Coding Theory*, 1998.

[Big73] N. Biggs. Perfect codes in graphs. *J. Combibatorial Theory(B)*, 15:289–296, 1973.

[BMvT78] E.R. Berlekamp, R.J. McEliece, and C.A van Tilborg. On the inherent intractability of certain coding problems. *IEEE Trans. Inf. Theory*, IT-24(3):384–386, May 1978.

[BN90] J. Bruck and M. Noar. The hardness of decoding linear codes with preprocessing. *IEEE Trans. Inf. Theory*, IT-36:331–335, 1990.

[CJ85] P. Cull and E.F. Ecklund Jr. Towers of Hanoi and Analysis of Algorithms. *American Mathematical Monthly*, 92(6):407–420, June-July 1985.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, 1979.

[Ham50] R. Hamming. Error detecting and error correcting codes. *Bell Syst. Tech. J.*, 29:147–160, 1950.

[Hil86] R. Hill. *A First Course in Coding Theory.* Oxford University Press, Oxford, 1986.

[KK88] J. Kratochvii and M. Krivanek. On the computational complexity of codes in graphs. *Lecture Notes in Computer Science*, 324:396–404, 1988.

[Kra87] J. Kratochvii. Perfect Codes in Graphs. *Proc. VII Hungarian Colloq. Combin. Eger*, 1987.

[Kra91] J. Kratochvii. *Perfect Codes in General Graphs.* Academia, Prague, 1991.

[Kra94] J. Kratochvii. Regular Codes in Regular Graphs are Difficult. *Discrete Mathematics*, 133:191–205, 1994.

[MS77] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes.* North-Holland, Amsterdam, 1977.

[NH81] S.C. Ntafos and S.L. Hakimi. On the complexity of some coding problems. *IEEE Trans. Inf. Theory*, IT-27(6):794–796, Nov 1981.

[TP71] A. Tietavainen and A. Perko. There are no unknown perfect binary codes. *Ann. Univ. Turku*, 148:3–10, 1971.