Perfect One Error Correcting Codes and Complete Iterated Graphs

Christopher Frayer
Grand Valley State University

Shalini Reddy
Santa Clara University

Advisor: Paul Cull
Oregon State University

8/15/02

Abstract

Given a graph is it possible to find a perfect one error correcting code on the graph or equivalently to find a perfect dominating set. The chosen vertices are called codevertices and have the properties that no two codevertices are adjacent and every noncodevertex is adjacent to exactly one codevertex. While the p1ecc problem is NP-complete for general graphs, the problem is easy for iterated complete graphs.

1 Introduction

In general coding on graphs is a difficult task; the existence of p1ecc on graphs has been shown to be NP complete, [1]. Cull and Nelson [1] developed codes on graphs that describe the Towers of Hanoi puzzle. A legal configuration of disks in the Towers of Hanoi puzzle is represented by a vertex, while a legal move of a disk is represented by an edge between two vertices. They found an unique perfect-one error correcting code on the graphs and a method for labeling that has finite state machines for codeword recognition and error correction, as well as easy methods for decoding and encoding. In 2001, Alspaugh, Knight, and Meloney, [2], generalized these results by generating perfect-one error correcting codes on n-iterations of complete graphs with d vertices given any natural numbers n and d. Alspaugh, Knight, and Meloney came up with three labeling methods that had finite state machines for codeword recognition, error correction, and encoding and decoding. We will introduce a labeling based on the $G-U$ construction so that there are finite state functions that map this labeling to all of the labelings devised by Alspaugh, Knight, and Meloney. We will also introduce another infinite family of labelings that have finite state characteristics. Then we will prove the existence of labelings that do not have finite state machines for codeword recognition and error correction, and provide an infinite family of such labelings.

1.1 Background and Definitions

This section gives definitions from graph theory and machine theory that we will use in the rest of the paper.

Definition 1.1 *A* ***graph*** *G consists of a nonempty finite set, $V(G)$, of elements called* ***vertices*** *and a finite set, $E(G)$, of distinct unordered pairs of distinct elements of $V(G)$ called* ***edges****.*

An example of a graph, G, is as follows. The graph G has the vertex set $V(G)=\{a, b, c\}$ and the edge set $E(G)=\{(a,b), (a,c), (b,c)\}$. Note that edges can also be denoted without the parenthesis, i.e. $E(G)=\{ab, ac, bc\}$.

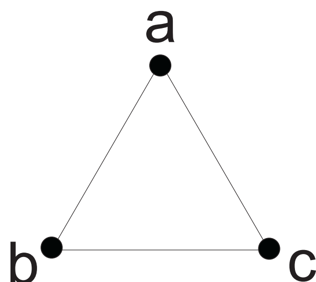

Figure 1: Example of a graph

The following is a list of some important characteristics of graphs.

Definition 1.2 *The* **degree** *of a vertex v_0, denoted $deg(v_0)$, is the number of adjacent vertices. (In the example above $deg(A)=deg(B)=deg(C)=2$.)*

Definition 1.3 *A* **subgraph** *M of a graph G consists of a subset $V(M) \subset V(G)$ together with the associated edges.*

Definition 1.4 *Two distinct subgraphs K and M of a graph G are* **adjacent** *if there exists $k \in K$ and $m \in M$ such that $km \in E(G)$.*

Definition 1.5 *A* **complete graph on d vertices** *is a graph in which each pair of distinct vertices is adjacent. We will denote the complete graph on d vertices by K_d.*

Figure 2 shows complete graphs on $d = 3$ and $d = 5$ vertices.

Figure 2: Complete graphs K_3 and K_5

The following is a description of the iterative construction for a bi-infinite family of graphs, K_d^n. The first iteration of the graph, K_d^1 is simply the complete graph on d-vertices. The second iteration, K_d^2, is constructed of d copies of K_d^1 and connected so that each copy is adjacent. In general, when constructing K_d^n, we create d copies of K_d^{n-1} and form edges between the corner vertices of the K_d^{n-1} such that any one of the d copies of K_d^{n-1} is adjacent to the other $d-1$ copies of K_d^{n-1}.

The process of constructing K_d^n consists of making d copies of $K_d^{n-1}$and forming edges between pairs of subgraphs such that an edge connects a corner vertex of one subgraph with a corner vertex of another subgraph so that every subgraph is adjacent to every other subgraph.

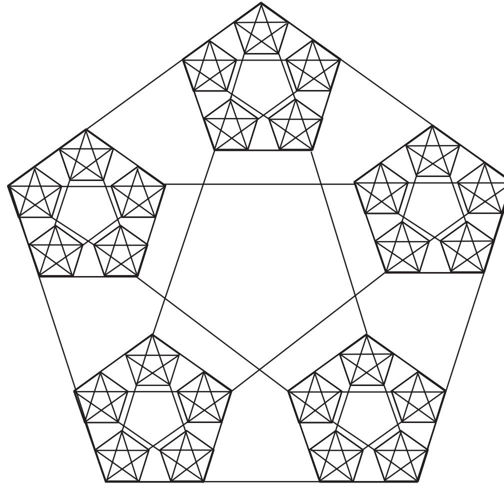

Figure 3: K_5^3

Definition 1.6 *A* **corner vertex** *is a vertex of an iterated complete graph K_d^n whose degree is $d-1$. An* **internal vertex** *is a non-corner vertex.*

Definition 1.7 *[6] A* **Finite state machines** *is defined by the quintuple*

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

Q is a finite set of internal states,

Σ is a finite set of symbols called the input alphabet,
$\delta : Q \times \Sigma \to Q$ is a total function called the transition function,
$q_0 \in Q$ is the initial state,
$F \subset Q$ is a set of final states.

2 Codes on Graphs

We now look at perfect one-error correcting codes on the bi-infinite family of complete iterated graphs. We will begin our description of codes on graphs with a few definitions.

2.1 Defintions

Definition 2.1 *A* **code** *on a graph G is any subset of vertices $C(G) \subset V(G)$. Vertices $c \in C(G)$ are called* **codevertices**. *Vertices $v \in V(G) - C(G)$ are called* **noncodevertices**.

From this definition we see that noncodevertices are simply those vertices not chosen as codevertices. When applying a coding scheme to a set of vertices, it is useful to associate each noncodevertex with a distinct codevertex. For instance, in a Hamming code every noncodevertex is within a distance d of a codevertex. The particular coding scheme that we will be looking at associates each noncodevertex to an adjacent codevertex. These codes are called perfect one-error correcting codes.

Definition 2.2 *A* **perfect one error correcting code**, *(which we will abbreviate as p1ecc) on a graph is a code that satisfies the following properties:*

1. *No two codevertices are adjacent*
2. *Every noncodevertex is adjacent to exactly one codevertex.*

For each iterated complete graph, (K_d^n), there is a unique p1ecc. The process of choosing these codewords occurs through the iterative construction of these graphs, which is made up of two different types of codes called G-codes and U-codes. These two codes are weak codes are defined in the following section.

2.2 Weak Codes and Unique p1ecc's on K_d^n

Weak codes are nearly p1ecc's. However, we allow some corner vertices to be noncodevertices, that are also not adjacent to a codevertex. We will prove the existence and uniqueness of weak codes on K_d^n, from which the existence and uniqueness of p1ecc's on K_d^n will directly follow. All of the definitions and theorems in this subsection are from the REU paper written by Alspaugh, Knight, and Meloney, [2].

Definition 2.3 *A* **weak code** *on a graph is a code that satisfies the following properties:*

1. *No two codevertices are adjacent*

2. *Every internal vertex is a codevertex or adjacent to exactly one codevertex.*

Definition 2.4 *If a vertex $v \in K_d^n$ is:*

1. *A codevertex that is not adjacent to any other codevertex, denote it by c.*

2. *A noncodevertex that is adjacent to exactly one codevertex, denote it by a.*

3. *A noncodevertex that is not adjacent to any codevertex, denote it by x.*

Definition 2.5 *A* **perfect one error correcting connection**, *(which we will abbreviate as p1ecc-c) is a connection between two corner vertices v_1 and v_2 of two different K_d^{n-1} subgraphs of K_d^n such that v_1 and v_2 satisfy the following properties after being connected:*

1. *No two codevertices are adjacent (v_1 and v_2 are not both codevertices)*

2. *For both v_1 and v_2, either v_i is a codevertex or is adjacent to exactly one codevertex.*

It is important to note that a p1ecc-c will preserve weak codes and p1ecc's.

Lemma 2.6 *There are exactly two p1ecc-c, they are x-c and a-a [2].*

Proof: There exists three types of vertices: x, a, and c. Therefore there are six possible connections: $x-a$, $x-c$, $a-c$, $a-a$, $x-x$, and $c-c$. We cannot have $c-c$ because no two codevertices can be adjacent. We cannot have $x-x$ because both x vertices will not be adjacent to a codevertex. We cannot have a $c-a$ because a will be adjacent to two codewords. We cannot have $x-a$ because the x vertex will not be adjacent to any codevertex. So there is at most two connections: $a-a$ and $x-c$. To see that there are p1ecc-c's, note that for $a-a$ clearly no two codewords are adjacent and each vertex is adjacent to a codevertex c. Note that with a $x-c$ connection, an x vertex is adjacent to exactly one codevertex. It is clear that no two codevertices are adjacent, thus $x-c$ is piecc-c.

■

The following definitions describe the two types of weak codes, which will we use to prove that the construction method for choosing codevertices on K_n^d is unique.

Definition 2.7 *A G-code, G_d^n, is a weak code on d-vertices of n-iterations such that if*

1. *n is even, then all corner vertices are c.*

2. *n is odd, then exactly one corner vertex is a c, and all other corner vertices are a.*

Definition 2.8 *A U-code, U_d^n, is a weak code on d-vertices of n-iterations such that if*

1. *n is even, exactly one corner vertex is an x, and all the other corner vertices are a.*

2. *n is odd, then all of the corner vertices are x.*

Theorem 2.9 *There are exactly two weak codes on K_d^n, the G-code and the U-code [2].*

Proof: We will use induction on n, the number of iterations. For $n = 1$, K_d^1 has exactly two weak codes. If no vertex is chosen as a codeword, then K_d^1 is a U-code. If a single vertex is chosen as a codeword, then K_d^1 is a G-code.

For our induction, we assume that for $n = k - 1$ there exists exactly two weak codes, namely G_d^{k-1} and U_d^{k-1}, then there are two cases to consider. Now we must consider two cases.

Case 1: K_d^k has at least one corner vertex that is a c.

If a corner vertex of K_d^k is a c, then it must come from some copy of G_d^{k-1}, since U_d^{k-1} has no c corner vertices. We must now consider the two subcases of k odd and k even.

If k is even, then the other $d - 1$ corner vertices in G_d^{k-1} are a by the definition of a G-code. Since the only p1ecc-c are $x - c$ and $a - a$, the a corner vertices can only connect to other a corner vertices. However, by the definition of a U-code every corner vertex of U_d^{k-1} will be an x. Therefore no copy of U_d^{k-1} will appear in K_d^k. Since each a in G_d^{k-1} can only connect to another, we will have d-copies of G_d^{k-1}. By connecting G_d^{k-1} subgraphs with $a - a$, K_d^k is a weak code. Since the c corner vertex of each G_d^{k-1} remains unconnected, each corner vertex of K_d^k is a c, thus K_d^k is actually G_d^k.

If k is odd, then by definition all the other corner vertices of G_d^{k-1} are c. Since we know that all the corner vertices of G_d^{k-1} are c, then the other $d - 1$ subgraphs must have an x corner vertex, which will come from $d - 1$ copies of U_d^{k-1}. Then $d - 1$ of the corner vertices of G_d^{k-1} will make $x - c$ connections with the $d - 1$ U_d^{k-1} subgraphs. Thus K_d^k is a weak code. Furthermore, since one corner vertex of K_d^k is a c corner vertex from G_d^{k-1}, and the other $d - 1$ corner vertices are a corner vertices from U_d^{k-1}, K_d^k is actually G_d^k.

We have just shown that in the case where at least one of the corner vertices of K_d^k is a c, then there is exactly one weak code that is produced, a G-code.

In this case, we see that if at least one corner vertex of K_d^k is a c, then either all the corner vertices are c or exactly one corner vertex is a c. Thus the only other choice left to consider is if K_d^k has no corner vertex that is a c.

Case 2: No corner vertex of K_d^k is a c.

Let no corner vertex of K_d^k be of type c, then K_d^k can not be constructed of only G_d^{k-1} subgraphs. This is because the corner vertices of G_d^{k-1} are either c or a, and by our assumption, we can not have any corner vertex of K_d^k be a c. If K_d^k was constructed of only G_d^{k-1}, it may necessarily force an illegal $a - c$ connection. Therefore, we can conclude that K_d^k must be connected of at least one U_d^{k-1}. We will now consider the cases of k odd and k even.

If k is even then all corner vertices of U_d^{k-1} are x by the definition of a U-code. Since the only p1ecc-c's for an x vertex is an $x - c$ connection and G_d^{k-1} is the only code that has c as a corner vertex, we must connect U_d^{k-1} to a G_d^{k-1}, which has exactly one c as a corner vertex. Therefore $d - 1$ of the x vertices in U_d^{k-1} will make $x - c$ connections with the $d - 1$ copies of G_d^{k-1}, and the G_d^{k-1} will make $a - a$ connections amongst themselves. Furthermore, we see that exactly one corner vertex of K_d^k will be x and the remaining $d - 1$ corner vertices will be a. So we see that K_d^k is in fact the weak code U_d^k.

If k is odd then U_d^{k-1} has exactly one x as a corner vertex, and the remaining corner vertices must be a. We see that this x vertex must be a corner vertex of K_d^k, otherwise it

would be connected to a c, which belongs only to a G_d^{k-1}. If we had a G_d^{k-1}, all of whose corner vertices are c, then at least one corner vertex of K_d^k would be a c which is a contradiction to our assumption. Thus, we can only use U_d^{k-1}. The other $d-1$ vertices of U_d^{k-1} are all a and can only be connected to another a corner vertex of a U_d^{k-1}. Furthermore, the x vertex in each of the U_d^{k-1} will become the corner vertex of U_d^k and remain unconnected. Hence K_d^k will have a weak code, namely U_d^k.

In this case we have seen that if no corner vertex of K_d^k is a c, then exactly one weak code is produced on K_d^k, namely a U-code. Thus by induction there are exactly two weak codes on K_d^n, U-codes and G-codes.

■

We have just shown the proof of existence and uniqueness of weak codes, from which the existence and uniqueness of p1ecc'c on K_d^k will easily follow. Some K_d^n graphs are made out of d copies of the same K_d^{n-1} subgraphs. Therefore K_d^n is isomorphic to any rotated copy of itself. As part of the uniqueness argument we will need to distinguish between these isomorphic graphs. To do this we will fix a top vertex of K_d^n, which allows us to distinguish between isomorphic graphs.

Theorem 2.10 *There is exactly one (up to isomorphism) p1ecc on K_d^n[2].*

Proof: The p1ecc on K_d^n is the G-code. When n is even the G-code will be invariant under rotations of the graph, and is truly unique. However, when n is odd, only one corner vertex will be a codevertex. Thus the unique p1ecc is found after fixing the codevertex as the top vertex. Therefore we conclude that there is exactly one (up to isomorphism) p1ecc on K_d^n.

■

2.3 The G-U Construction

Throughout this explanation we will illustrate the process with the $d = 3$ case (triangles). However, it works for any natural number d, and we will describe the construction generally for any choice d and n. First we will construct two complete graphs on d-vertices. For G_d^1, pick one vertex as a codevertex and rotate the codevertex to the top position. We will now refer to this vertex as the **top vertex**. All of the other vertices will be referred to as nontop corner vertices. In the construction of U_d^1 no vertex is picked as a codevertex, instead we pick one vertex to be the top vertex. Figure 4 illustrates G_3^1 and U_3^1. The code vertex in G_3^1 is labelled with a black dot.

Figure 4: $n = 1$ G-U construction

Now we will describe the construction for any value $n > 1$. We will describe both G_d^n and U_d^n in a logical iterative fashion.

We first look at G_d^n when n is even. Create d-copies of G_d^{n-1} and connect them such that:

1. The top vertex of each copy of G_d^{n-1} remains unconnected.
2. Each copy is adjacent.
3. Let the top vertex of some G_d^{n-1} be the top vertex of G_d^n.

We see that all the corner vertices are c vertices.

We now will look at U_d^n when n is even. Create 1-copy of U_d^{n-1} and $d-1$ copies of G_d^{n-1} and connect them such that:

1. The top vertex of each copy of G_d^{n-1} is connected to a distinct nontop vertex of U_d^{n-1}.
2. Connect nontop corner vertices of G_d^{n-1} to the other $d-2$ copies of G_d^{n-1} such that they are all adjacent and one corner vertex remains unconnected.
3. The top vertex of U_d^{n-1} becomes the top vertex of U_d^n.

We see that the top vertex will be an x, and the remaining $d-1$ vertices will be a.

For G_d^n, where n is odd. Create 1 copy of G_d^{n-1} and $d-1$ copies of U_d^{n-1} and connect them such that:

1. The top vertex of each copy of U_d^{n-1} is connected to a distinct nontop vertex of G_d^{n-1}.
2. Connect nontop corner vertices of U_d^{n-1} to the other $d-2$ copies of U_d^{n-1} such that they are all adjacent and one nontop corner vertex remains unconnected.
3. The top vertex of G_d^{n-1} becomes the top vertex of G_d^n.

We see that the top vertex will be a c, and the remaining $d-1$ vertices will be a.

For U_d^n where n is odd. Create d-copies of U_d^{n-1} and connect them such that:

1. The top vertex of each copy of U_d^{n-1} remains unconnected.
2. Each copy is adjacent.
3. Let one of the top vertices of some U_d^{n-1} be the top vertex of U_d^n.

We see that all the corner vertices are x vertices.

Figures 5 and 6 show G_3^2, U_3^2, G_3^3, and U_3^3. These figures along with the above explanations should give the reader an intuitive as well as a pictorial sense of the G-U construction process.

Figure 5: $n = 2$ G-U construction

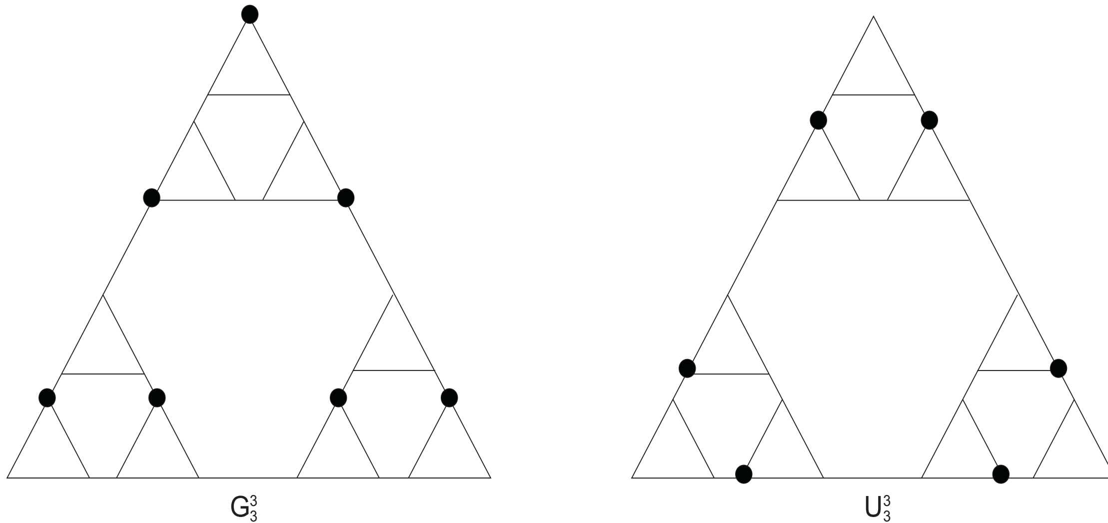

Figure 6: $n = 3$ G-U construction

3 Labeling Methods of K_d^n

During the Oregon State University Summer 2001 REU program Alspaugh, Knight, and Meloney came up with three distinct labeling methods for K_d^n, [2]. These methods and their properties are explained in this section.

3.1 The C-R-E-L Labeling

The C-R-E-L labeling is an easy to follow labeling method. This labeling method follows exactly from the *G-U* construction. This method has a finite state machine that acts as codeword recognizer and another finite state machine that acts as a noncodeword sorter so that words can be error corrected.

The C-R-E-L labeling for a K_d^n graph consists of strings of length n taken over the alphabet $\{0, 1, \cdots, d-1\}$. This subsection explains how to connect the d copies of K_d^{n-1}

$(C_0, C_1, \cdots, C_{d-1})$ by specifying how to connect corner vertices. Throughout the labeling construction we will prefix a character to the left of a string at each iterative step, this process will be called prefixing.

For G_d^1 label the codevertex, which is also referred to as the top vertex, 0 and the remaining $d-1$ vertices with a distinct $i \in \{1, 2, \ldots, d-1\}$. For U_d^1 label the top vertex, 0 and the remaining $d-1$ vertices with a distinct $i \in \{1, 2, \ldots, d-1\}$. This can be seen in figure 7 with the d=3 case.

Figure 7: K_3^1 C-R-E-L labeling

Each K_d^n is constructed from d copies of K_d^{n-1}, which we call $C_0, C_1, \cdots, C_{d-1}$. For each $i \neq j$, we form exactly one edge between c_i and c_j, where c_i and c_j are labels of length $n-1$ of corner vertices in C_i and C_j, respectively, such that one of the following two construction rules is satisfied:

The Congruence Connection Rule:

1. The leftmost character of c_i is k where $k \equiv j - i \,(mod\, d)$.
2. The leftmost character of c_j is l where $l \equiv i - j \,(mod\, d)$.

The Swap Connection Rule:

1. The leftmost character of c_i is j.
2. The leftmost character of c_j is i.

After making these connections, each vertex in C_i is prefixed with i. Figures 8 and 9 illustrate each of the two connection rules.

We are now ready to explicitly describe how to construct and label G_d^n and U_d^n. The congruence connection rule is used for constructing G_d^n for n even and U_d^n for n odd. The swap connection rule is used for constructing G_d^n for n odd and U_d^n for n even.

To construct G_d^n for n even:

1. Create d labelled copies of G_d^{n-1}, call these $C_0, C_1, \cdots, C_{d-1}$

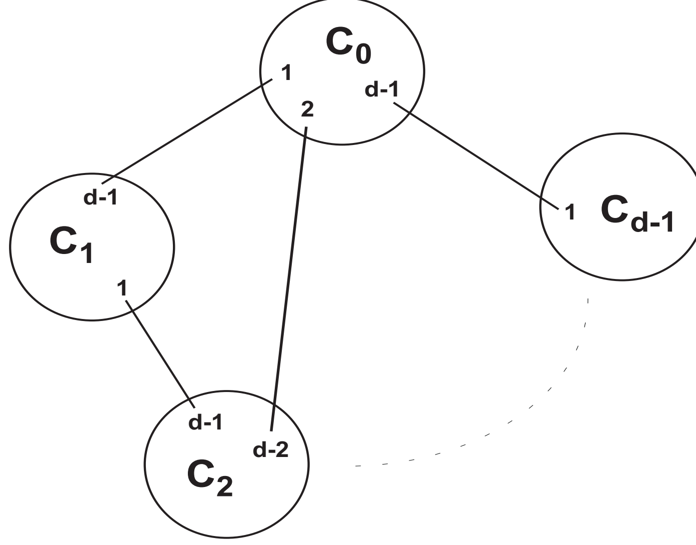

Figure 8: Congruence connection rule

2. Designate the top vertex of C_0 as the top vertex of G_d^n.
3. For $i \neq j$, form one edge between c_i and c_j according to the **congruence connection rule**.
4. For every vertex $x_i \in C_i$ prefix x_i with an i.

To construct U_d^n for n odd:

1. Create d labelled copies of U_d^{n-1}, call these $C_0, C_1, \cdots, C_{d-1}$
2. Designate the top vertex of C_0 as the top vertex of U_d^n.
3. For $i \neq j$, form one edge between c_i and c_j according to the **congruence connection rule**.
4. For every vertex $x_i \in C_i$ prefix x_i with an i.

To construct G_d^n for n odd:

1. Create one labelled copy of G_d^{n-1}, call this C_0, and designate the top vertex of C_0 as the top vertex of G_d^n.
2. Create $d-1$ labelled copies of U_d^{n-1}, call these $C_1, C_2, \cdots, C_{d-1}$.
3. For $i \neq j$, form one edge between c_i and c_j according to the **swap connection rule**.
4. For every vertex $x_i \in C_i$ prefix x_i with an i.

To construct U_d^n for n even:

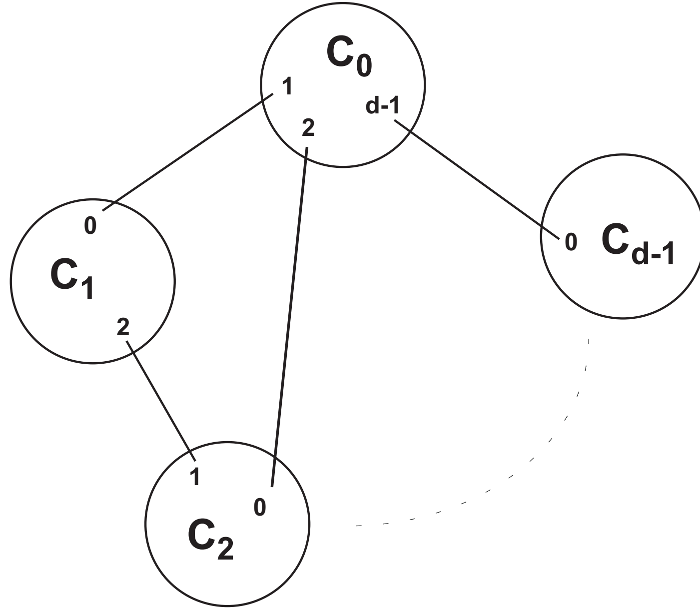

Figure 9: Swap connection rule

1. Create one labelled copy of U_d^{n-1}, call this C_0, and designate the top vertex of C_0 as the top vertex of U_d^n.
2. Create $d-1$ labelled copies of G_d^{n-1}, call these C_1, C_2, $\cdots$, C_{d-1}.
3. For $i \neq j$, form one edge between c_i and c_j according to the **swap connection rule**.
4. For every vertex $x_i \in C_i$ prefix x_i with an i.

Note that this labeling method is based on the $G-U$ construction which is described in the previous section. Figures 10 and 11 show the G_3^2, G_3^3, U_3^2, and U_3^3 graphs.

Figure 10: K_3^2 C-R-E-L labeling

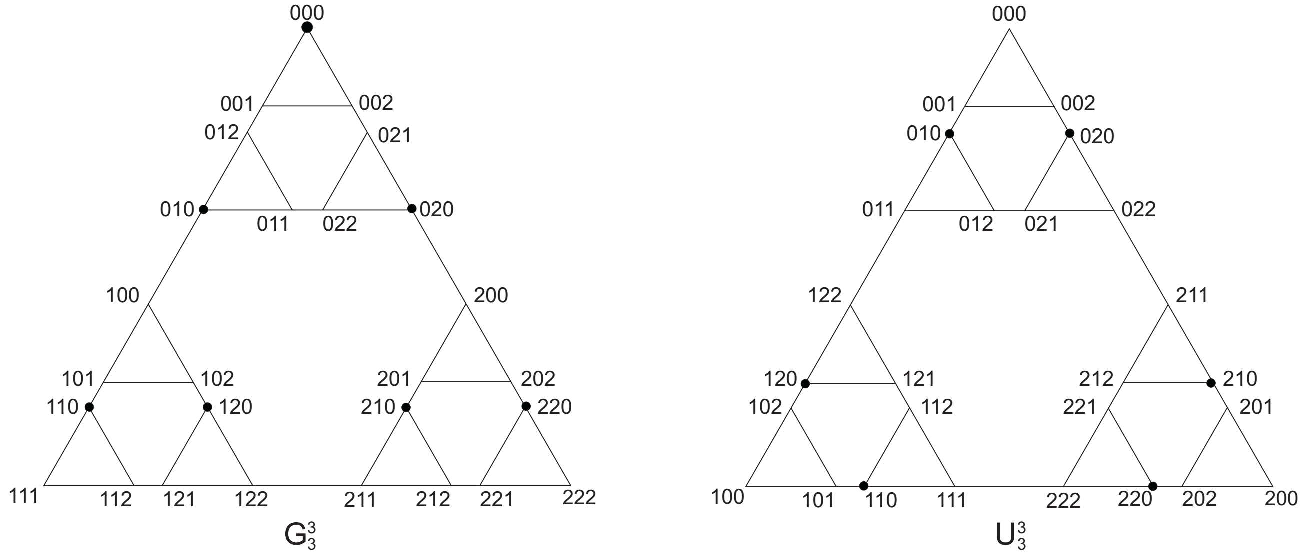

Figure 11: K_3^3 C-R-E-L labeling

3.2 α-Method

The α-method labeling is a form of Gray code labeling.

Definition 3.1 *A* **Gray code** *is a labeling with the property that there is a difference of one character between the labels of two adjacent vertices.*

The α-method has a finite state machine that acts as a codeword recognizer and another finite state machine that acts as an error corrector.

To discuss Gray code labeling of K_d^n it is necessary to discuss the ordering of vertices. It is important to note that labeling is different from labeling and should be treated differently. We will now describe ordering.

For $n = 1$, K_d^1, the top vertex is ordered 0 and each following corner vertex (moving counterclockwise) is ordered with the successive integer through $d-1$. Given the K_d^{n-1} construction, K_d^n is made by constructing d copies of K_d^{n-1} subgraphs (ordered zero through $d-1$) such that the jth corner vertex of the kth copy of K_d^{n-1} connects to the kth corner of the jth copy of K_d^{n-1}. The vertices for the complete graph K_d^1 are labelled identically as they are ordered. Figure 12 gives an example of the labelled figure K_3^1.

Given an arbitrary value of n, to construct K_d^n we make d copies of labelled K_d^{n-1}. K_d^n is labelled by rotating the kth subgraph of a labelled K_d^{n-1} $\frac{360*k}{d}$ degrees clockwise. Then the

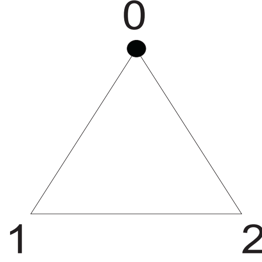

Figure 12: K_3^1 α-method labeling

character k is prefixed to the front of the labeling of K_d^{n-1} for each vertex in that subgraph. Figures 13 and 14 show the α-method labeling for $n = 2$ and $n = 3$ respectively.

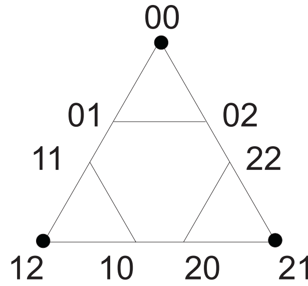

Figure 13: K_3^2 α-method labeling

3.3 Multiples of d+1 code

The multiples of $d + 1$ code, which will be referred to as mod$(d + 1)$ code, is a labeling method for K_d^n that has finite state machines for codeword recognition, error correction, as well as easy methods for encoding and decoding.

We will first explain the mod$(d + 1)$ labeling for $n = 1$, and then we will explain it for arbitrary n. For $n = 1$, K_d^1, the top vertex is labelled 0 and each following corner vertex (moving counterclockwise) is labelled with successive integers through $d - 1$. Figure 15 gives an example of the labelled figure K_d^1.

To construct K_d^n we create d copies of K_d^{n-1} then connect the respective subgraphs C_0, C_1, …, C_d using the swap connection method. Next we prefix every string within the C_k subgraph with k, where $k \in \{0, \ldots, d - 1\}$. Then C_k (subgraph number k) will be *position swapped* according to the following rule:

We will position swap subgraphs i and j within C_k, for all i and j such that $i + j \equiv d + k$ mod$(d + 1)$. (Note that for any given $j \in 0, 1, \ldots, d - 1$ satisfies the condition with exactly one i, unless $j = k$, in which case it satisfies the position swapping with no i.) We will now formally define the position swapping isomorphism.

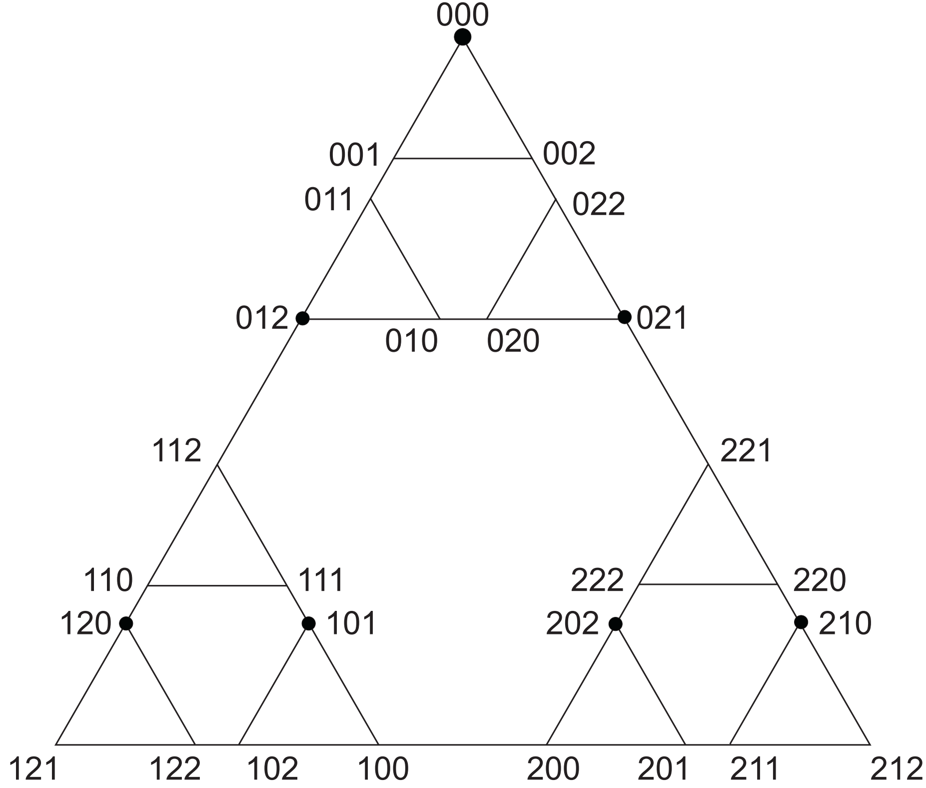

Figure 14: K_3^3 α-method labeling

Definition 3.2 *When applying position swapping to vertices in K_d^n, the vertices in the ith and j^{th} positions within each K_d^{n-1} subgraph switch places, and then the vertices in the ith and j^{th} positions within each K_d^{n-2} sub-subgraph switch places, ..., down to the vertices in the ith and j^{th} of every K_d^1 base graph.*

Figures 16 and 17 show the mod$(d+1)$ labeling for K_3^2 and K_3^3 respectively.

4 C-S Labeling Method

This year, we came up with a more general labeling that is based on the $G-U$ construction. It is much different from the C-R-E-L, α-method, and mod$(d+1)$ labeling methods in that the C-S alphabet consists of $5d-2$ characters: G_0^0, ..., G_{d-1}^0, G_1^1, ..., G_{d-1}^{d-1}, U_0^0, ..., U_{d-1}^0, U_1^1, ..., U_{d-1}^{d-1}, 0, ..., $d-1$. Given the subscript i on G or U, where $i \in 0,\ldots,d-1$, denotes the subgraph position. Given the superscript i on G or U, where $i \in \{0, \ldots, d-1\}$, the degree of rotation is equal to $\frac{360*i}{d}$.

The C-S labeling for any K_d^n graph consists of strings of length n taken over the C-S alphabet. For G_d^1 label the codevertex, which is also the top vertex, 0 and the remaining $d-1$ with a distinct $i \in\{1, 2, \ldots, d-1\}$. For U_d^1 label the top vertex, 0 and the remaining $d-1$ vertices with a distinct $i \in\{1, 2, \ldots, d-1\}$. This is illustrated in figure 18 for d=3.

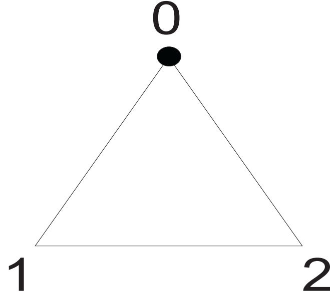

Figure 15: K_3^1 mod$(d+1)$ labeling

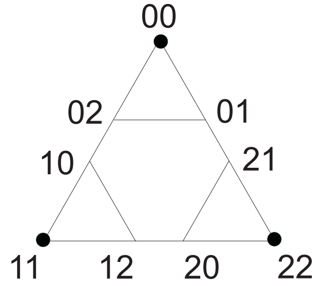

Figure 16: K_3^2 mod$(d+1)$ labeling

The congruence connection rule is used for constructing G_d^n for n-even and U_d^n for n-odd, because it accounts for the rotations that occur. The swap connection rule is used for constructing G_d^n for n-odd and U_d^n for n-even, since no rotations occur. We consider the subscripts of G and U when using the congruence connection rule and the swap connection rule

To construct G_d^n for n even:

1. Create d labelled copies of G_d^{n-1}, call these $C_0, C_1, \cdots, C_{d-1}$.
2. Designate the top vertex of C_0 as the top vertex of G_d^n.
3. For $i \neq j$, form one edge between c_i and c_j according to the **congruence connection rule**.
4. For every vertex $x_i \in C_i$ prefix x_i with an G_i^i.

To construct U_d^n for n odd:

1. Create d labelled copies of U_d^{n-1}, call these $C_0, C_1, \cdots, C_{d-1}$.
2. Designate the top vertex of C_0 as the top vertex of U_d^n.
3. For $i \neq j$, form one edge between c_i and c_j according to the **congruence connection rule**.

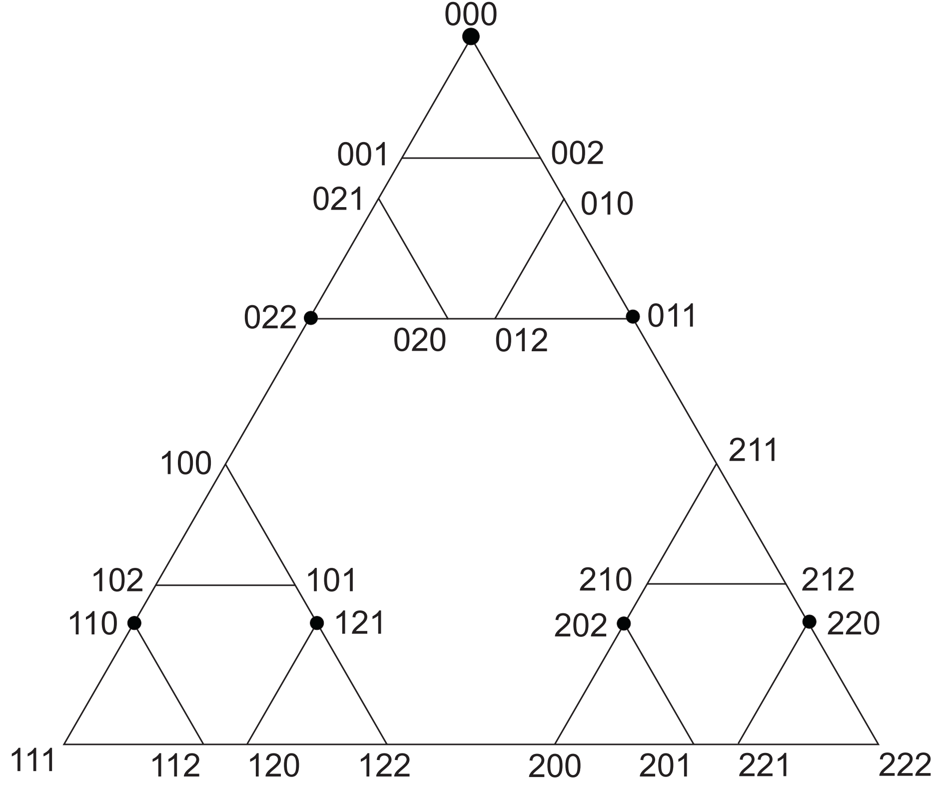

Figure 17: K_3^3 mod$(d+1)$ labeling

4. For every vertex $x_i \in C_i$ prefix x_i with an U_i^i.

To construct G_d^n for n odd:

1. Create 1 labelled copy of G_d^{n-1}, call this C_0, and designate the top vertex of C_0 as the top vertex of G_d^n.
2. Create $d-1$ labelled copies of U_d^{n-1}, call these C_1, C_2, $\cdots$, C_{d-1}.
3. For $i \neq j$, form one edge between c_i and c_j according to the **swap connection rule**.
4. For every vertex $x_0 \in C_0$ prefix x_0 with an G_0^0.
5. For every vertex $x_i \in C_i$ where $i \in \{1, 2, \ldots, d-1\}$, prefix x_i with an U_0^i.

To construct U_d^n for n even:

1. Create 1 labelled copy of U_d^{n-1}, call this C_0, and designate the top vertex of C_0 as the top vertex of U_d^n.
2. Create $d-1$ labelled copies of G_d^{n-1}, call these C_1, C_2, $\cdots$, C_{d-1}.

Figure 18: K_3^1 for CS labeling

3. For $i \neq j$, form one edge between c_i and c_j according to the **swap connection rule**.
4. For every vertex $x_0 \in C_0$ prefix x_0 with an U_0^0.
5. For every vertex $x_i \in C_i$ where $i \in \{1, 2, \ldots, d-1\}$, prefix x_i with an G_0^i.

Figures 19 and 20 show the G_3^2, G_3^3, U_3^2, and U_3^3 graphs.

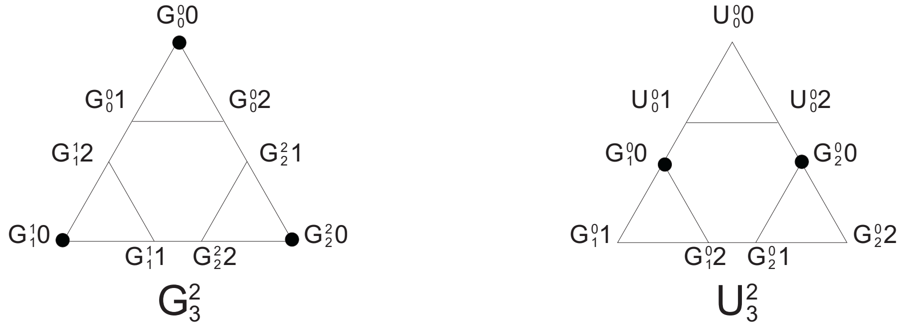

Figure 19: K_3^2 for CS labeling

5 C-S Mappings

Within this section we show mappings from the C-S labeling method to the C-R-E-L and α-method labeling method using a finite state function and to the mod$(d+1)$ labeling methods using a non-finite state function. First we will define a finite state function.

Definition 5.1 *A* **Finite state machines** *is defined by the sextuple*

$$M = (Q, \Sigma, , \Sigma', \delta, \lambda, q_0)$$

where

Q is a finite set of internal states,
Σ is a finite set of symbols called the input alphabet,
Σ is a finite set of symbols called the output alphabet,
$\delta : Q \times \Sigma \rightarrow Q$ is a total function called the transition function,
$\lambda : Q \times \Sigma \rightarrow \Sigma'$ is a total function called the output function,
$q_0 \in Q$ is the initial state,

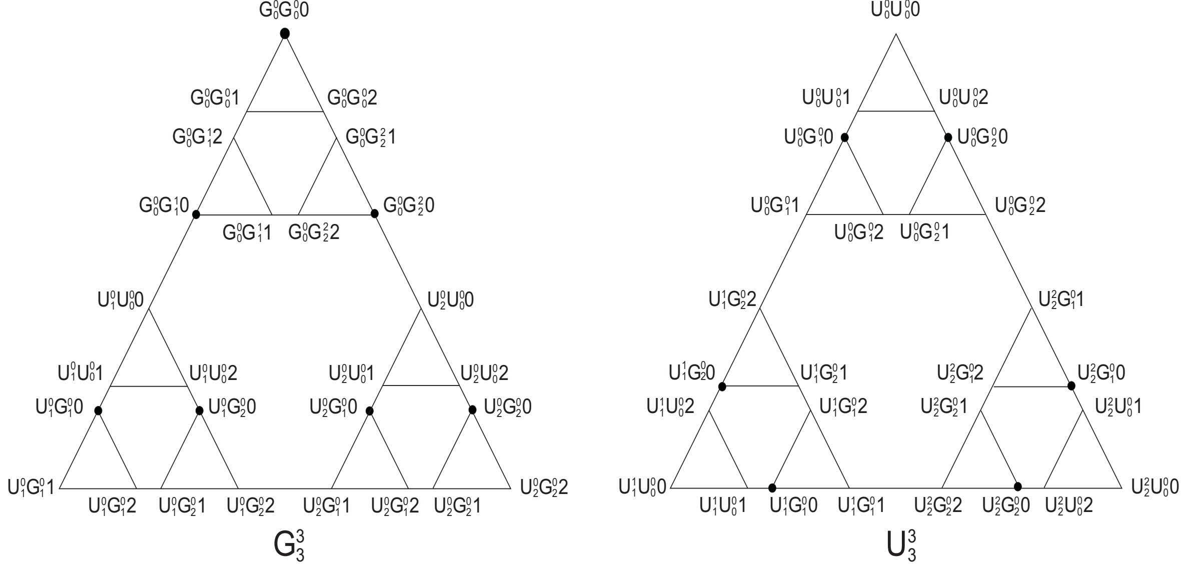

Figure 20:K_3^3 for CS labeling

5.1 C-S to C-R-E-L Finite State Function

The mapping from the C-S labeling to the C-R-E-L labeling is quite simple and is done using a finite state function. The finite state function has zero states, and consists only of a λ function. To define this function we will let L represent U or G, since the graph type does not effect the mapping.

Mapping 5.2 *The mapping from the C-S labeling to the C-R-E-L labeling is done by the finite state function consisting of no states and the following λ function, where $a, b \in \{0, \ldots, d-1\}$:*

$$\lambda(L_a^b) = a \textit{ and } \lambda(a) = a$$

We will now prove that this finite state function is a mapping from the C-S labeling to the C-R-E-L labeling.

Theorem 5.3 *The C-S to C-R-E-L finite state function is a mapping from the C-S labeling to the C-R-E-L labeling.*

Proof: We will use induction on the length of the string being mapped. The base case is trivial by construction.

We assume that for $n = k$ the finite state function defines a mapping from the C-S labeling to the C-R-E-L labeling. We will show that for $n = k + 1$ the finite state function defines such a mapping. Let $n = k + 1$. When $n = k$ the finite state function defines a proper mapping, (from C-S labeling to C-R-E-L labeling) we are only concerned about what happens to the left most character of a codeword. The leftmost character of the mapped C-S label will be the subgraph number of the C-S character, which by the definition of the

C-S construction is the same as the C-R-E-L leftmost character. Thus for $n = k + 1$ the finite state function defines a mapping from the C-S labeling to the C-R-E-L labeling.

■

5.2 C-S to α-method Finite State Function

The mapping from the C-S labeling to the α-method labeling is more complicated than the previous mapping. This finite state function has states and consists of both λ and δ functions. To define this function we will let L represent U or G, since the graph type does not effect the mapping.

Mapping 5.4 *The mapping from the C-S labeling to the α-method labeling is done by the finite state function with d states ($q_0, q_1, \ldots, q_{d-1}$) and the following λ and δ functions, where $a, b, c \in \{0, \ldots, d-1\}$:*

$$\lambda(a, q_b) = (a + b)\text{mod}d \text{ and } \lambda(L_a^b, q_c) = (a + c)\text{mod}d$$

$$\delta(L_a^b, q_c) = (a + b + 2c)\text{mod}d.$$

Where the initial state is q_0 and the strings are read from left to right.

From the definition of the finite state function, we see that a given character in the C-S string is mapped to the number which is equivalent to the subscript of the C-S character, plus the sum of the previous rotations, plus the sum of the already determined α-method characters taken mod(d). (If this is not apparent, working through several examples make this clear) In order to show that this finite state function is a mapping from the C-S labeling to the α-method, we will show that the subscript of the C-S character, plus the sum of the previous rotations, plus the sum of the already determined α-method characters taken mod(d) is the α character that should be mapped to. Before we are able to prove that this finite state function is a mapping from the C-S labeling to the α-method labeling we first must prove several lemmas.

Lemma 5.5 *The actual position of a subgraph within the next larger subgraph (the actual sub-...-($n-1$ times)-subgraph within the sub-...-($n-2$ times)-subgraph) of the nth character from the left of a C-S labeling method string is the subgraph of that character plus the number of previous rotations (all of the rotations in the first $n-1$ characters from the left of the string) taken mod(d).*

Proof: Let L_a^b be a character in a C-S string that is the nth character from the left of the string (If this digit is a constant, the last digit of the string, then simply make L_a^b into an a). If the previous number of rotations is zero, then L_a^b has not been rotated and the actual position of a subgraph within the next larger subgraph is position a.

Assume that the previous number of rotations is a number β such that $\beta \neq 0$. The L_a^b was originally in the a subgraph position, and then was rotated β times counterclockwise. Since the C-S labeling scheme numbers counterclockwise, $(a + \beta)$mod(d) gives the actual position of a subgraph within the next larger subgraph.

■

Lemma 5.6 *The nth character from the left of an α-method string is equivalent to the actual position of a subgraph within the next larger subgraph plus the sum of the first $n-1$ characters from the left of the α string mod(d).*

Proof: Consider an α-method string S, such that the nth character from the left is in the hth position of a subgraph within the next larger subgraph. Also, we will assume that the sum of the first $n-1$ characters form the string S (from the left) is m. The number m gives the number of rotations that has occurred clockwise from the left. The number $(h+m)$mod(d) (adding goes counterclockwise) gives us the graph number, that when rotated m times clockwise is in the hth position of a subgraph within the next larger subgraph. Thus the lemma is true.

■

Theorem 5.7 *The C-S to α-method finite state function is a mapping from the C-S labeling to the α-method labeling.*

Proof: To show that the C-S to α-method finite state function is a mapping from the C-S labeling to the α-method labeling we will show that the subscript of the C-S character, plus the sum of the previous rotations, plus the sum of the predetermined α-method characters taken mod(d) is the α character that should be mapped to. Let S be a C-S labeling string, such that the nth character from the left is L_h^a where h and a are constants, the sum of the previous rotations is x, and the sum of the already determined α-method characters is z. From lemma 5.5 we know that $(h+x)$mod(d) gives us the actual position of a subgraph within the next larger subgraph. Then lemma 5.6 tells us that $(h+x)+z$mod (d) is the α-method character that L_h^a should be mapped to. Therefore the C-S to α-method finite state function is a mapping from the $C-S$ labeling to the α-method labeling.

■

5.3 C-S to mod($d+1$) Function With a Counter

The mapping from the C-S labeling to the (mod $d+1$) labeling is more complex than the previous mapping. This mapping is not finite state, because it requires a counter. To define this function we will let L represent U or G, (whether a vertex is from a U or G graph does not affect this mapping). Before we explain the mapping we must first introduce the following functions:

$R(a)$ is a function that adds the previous number of rotations to a constant input mod(d).

q_a is a composition of functions

$f^{s(k)}$ is a function that given an input of L_a^b or a constant c changes the subgraph number a or the constant input c to what the isomorphism swapping rule changes an a or c to in the k subgraph.

Mapping 5.8 *The mapping from the C-S labeling to the mod($d+1$) labeling is done using a function with a counter using the following λ and δ functions, where $a, b, c \in \{0, \ldots, d-1\}$:*

$$\lambda(a,\, q_c) = q_c(R(a)) \text{ and } \lambda(L_a^b,\, q_c) = q_c(L_a^b)$$

$$\delta(L_a^b,\, q_c) = f^{sq_a(n)}(q_a)$$

Note that the counter in this function is q, because q keeps track of a finite number of function compositions. This function reads CS strings from left to right.

We will now prove that this function with the counter is a mapping from the CS method to the mod$(d+1)$ method.

Theorem 5.9 *The CS to mod($d+1$) function with a counter is a mapping from the CS labeling to the mod($d+1$) labeling.*

Proof: Consider a CS labelled string of length k. The first character of this string will be mapped to the subgraph number, which is the proper character for the corresponding mod$(d+1)$ labelled string in the first position. Now consider the nth character from the left of the CS string where $n < k$. The subscript of the nth character denotes the actual subgraph position of the nth character within the next larger subgraph. The $n-1$ characters to the left of the nth character applied the opposite of what the isomorphism swapping rule would to the nth character. Thus, the finite state function with counter will map the nth CS character to the proper mod$(d+1)$ character. Now consider the last character of the CS string, in the mod$(d+1)$ labeling method, the last digit is the actual vertex number within that sub-...-subgraph with the isomorphism swapping rule applied to it by all the characters to the left of it. Thus, by lemma 5.5, the function R applied to the last constant will give the actual position of the vertex within that sub-...-subgraph. Therefore the lambda function applied to this last character will give the proper last mod$(d+1)$ character.

Therefore he CS to mod$(d+1)$ function with a counter is a mapping from the CS labeling to the mod$(d+1)$ labeling.

■

6 Finite State Properties of CS labeling

An important fact concerning finite state functions and machines is as follows:

Lemma 6.1 *Let A represent a labeling that has a finite state machine for codeword recognition. If there is a finite state mapping from a labeling method B to this labeling method A, then B has a finite state machine for codeword recognition.*

The proof of this lemma comes directly from properties of finite state machines and finite state functions.

Theorem 6.2 *The CS labeling method has a finite state machine for codeword recognition.*

Proof: We have seen in theorem 5.7 that there is a finite state mapping from the CS labeling to the α-method labeling. The α-method labeling has finite state machines for error correction and codeword recognition, [2]. Thus lemma 6.1 tells us that the processes of codeword recognition is finite state in the CS labeling method.

■

The proof could also have used the finite state function from the CS to C-R-E-L labeling to prove the theorem.

7 Families of Labelings Based Upon the Symmetric Group

This section will introduce several labelings based upon symmetries that exist in the symmetric group.

7.1 Finite State Properties of Labelings Based on the Symmetries of the Regular d-gon

This subsection will discuss labelings that are based on the basic symmetries of the regular d-gon. By basic symmetries we mean the permutations of the regular d-gon that are in the dihedral group of that d-gon.

Definition 7.1 *Let P_n be the regular n-gon with n sides ($n \geq 3$). Number the vertices of P_n clockwise from 1 to n. Let D_n be the set containing all permutation of these vertices that can be obtained from rotating or reflecting the figure. The set D_n with the operation of function composition forms the nth dihedral group.*

Note that for $n = 2$ the only thing possible is a rotation of 180 degrees. This section will prove that any labeling based on using a basic symmetry has finite state machines for codeword recognition and error correction.

7.1.1 Finite State Mapping From Base Labeling to α-Method Labeling

This subsection will define and prove a finite state mapping from the base labeling to the α-method labeling method.

Definition 7.2 *The* **base labeling** *of a K_d^n graph is one that tells the actual subgraph position within the next larger subgraph; in other words, it gives the actual K_d^{n-1} subgraph position within K_d^n, for every character.*

A way to construct this labeling is to first pick a top vertex of K_d^1 and label it zero, and label the remaining $d-1$ vertices with a distinct $i \in \{1, 2, \ldots, d-1\}$.

To construct K_d^n, start with d labelled copies of $K_d^{n-1}$and call them C_0, C_1, C_2, $\cdots$, C_{d-1}. Then, for $i \neq j$, form one edge between c_i and c_j (the corner vertices of C_i and C_j respectively) according to the **swap connection rule**. Then for each vertex, x_i, such that x_i in C_i, prefix an i to x_i. Designate the top vertex of C_0 as the top vertex of K_d^n.

Figure 21 illustrates the base labeling for K_3^3.

Mapping 7.3 *The finite state mapping from the base labeling to the α-method labeling is done by the following λ and δ functions, where $a, b \in \{0, \ldots, d-1\}$:*

$$\lambda(a,\, q_b) = (a+b) \bmod d$$

$$\delta(a,\, q_b) = q_{(a+b) \bmod d}$$

This function reads base labelled strings from the left to right and starts in state q_0.

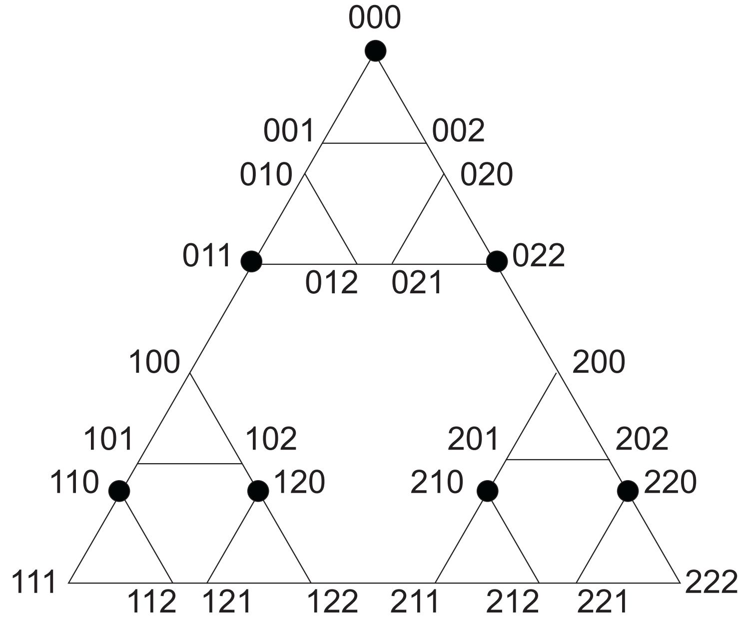

Figure 21: Base labeling of K_3^3

Theorem 7.4 *The base labeling to the α-method finite state function as defined above is a mapping from the base labeling to the α labeling.*

Proof: From the definition of the finite state machine we see that the output for a given character of a base labelled string is that character plus the sum of the previous α-method characters taken mod(d). Lemma 5.6 tells us that the nth character from the left of an α-method string is equivalent to the current subgraph position within the next larger subgraph plus the summation of the first $n-1$ characters from the left (of the α string) mod(d). Thus, lemma 5.6 proves the theorem.

■

Corollary 7.5 *The Base labeling method has a finite state machine for codeword recognition.*

Proof: We have seen that the α-method labeling has finite state machines for error correction and codeword recognition, [2]. Since there is a finite state mapping from the base labeling to the α-method labeling, lemma 6.1 tells us that the base labeling method has a finite state machines for codeword recognition.

■

7.1.2 Labelings based on a single reflection

This subsection will show that the processes of codeword recognition and error correction are finite state within labelings based on single reflections. At each iteration the subgraph is flipped about a single line of reflection ; however, only the subgraphs switch positions, and no reflections take place within the sub-...-subgraphs.

To construct this type of labeling first order the top vertex of K_d^1 as 0 and each following corner vertex (moving counterclockwise) with the successive integers through $d-1$. Given

the K_d^{n-1} construction, K_d^n is made by constructing d copies of K_d^{n-1} subgraphs (ordered zero through $d-1$) such that the jth corner vertex of the kth copy of K_d^{n-1} connects to the kth corner of the jth copy of K_d^{n-1}.

The vertices for the complete graph K_d^1 are labelled exactly as they are ordered. Then apply the single reflection. Given the K_d^{n-1} labeling, K_d^n is labelled by first prefixing the subgraph number to the left of each string. In K_d^1 the single reflection changes certain vertices. To finish the labeling of K_d^n switch the K_d^{n-1} subgraphs corresponding to the vertices switched in K_d^1. (For instance, if K_d^1 switches vertices 1 and 2, then K_d^n switches subgraphs 1 and 2.)

Mapping 7.6 *A finite state function from a labeling based on a single reflection to the base labeling has no states and the following λ functions:*
$\lambda(a) = a$ *if a is invariant under the single reflection, and*
$\lambda(a) = b$ *if a is changed to b under the single reflection.*
The strings are read from left to right.

Theorem 7.7 *The finite state function from a labeling based on single reflections to the base labeling as defined above is a mapping from the labeling based on a single reflection to the base labeling.*

Proof: Consider a string S from a labeling method based on single reflections. We will show that this string S is mapped to the proper base labelled string by the above finite state function. The first character is mapped exactly as described by the finite state function. If the first character has not been changed then it is mapped to that character. If the character has been changed, then it is mapped to the character it was changed from. We assume that the first $n-1$ characters from the left of the base string are properly converted. Given the nth character from the left we will show that this character is mapped to the proper base labelled character. If the nth character is such that it is invariant under the single reflection, then it remains unchanged. As is seen in the finite state function. However, if the nth character is such that it is changed under the single reflection it must be replaced with the character that it was changed from. This is also seen in the finite state function. Therefore in either case the nth character of the string S gets mapped to the proper character of the base labelled string.

■

Corollary 7.8 *A labeling method based on a single reflection has a finite state machines for codeword recognition.*

Proof: We have seen in corollary 7.5 that the base labeling has a finite state machines for codeword recognition. Theorem 7.7 showed that there is a finite state mapping from a labeling based on a single reflection to the base labeling. Thus lemma 6.1 tells us that a labeling method based on a single reflection has a finite state machines for codeword recognition.

■

7.1.3 Labelings Based on Single Rotations Within Each Subgraph

This subsection will show that the processes of codeword recognition and error correction are finite state within labelings based on single rotations within each subgraph. By single rotation within each subgraph, we mean that at each iteration each subgraph is rotated a certain number of degrees, $\frac{360*j}{d}$ degrees, where j is an integer modulos d.

A way to construct this type of labeling is to first order the top vertex of K_d^1 as 0 and each following corner vertex is ordered with the successive integer through $d-1$. Given the K_d^{n-1} construction, K_d^n is made by constructing d copies of K_d^{n-1} subgraphs (ordered zero through $d-1$) such that the jth corner vertex of the kth copy of K_d^{n-1} connects to the kth corner of the jth copy of K_d^{n-1}.

The labeling is done as follows. The vertices for the complete graph K_d^1 are labelled exactly as they are ordered. Given the K_d^{n-1} labeling, K_d^n is labelled by first prefixing the subgraph number to the left of each string. To finish the labeling of K_d^n rotate each subgraph the proper number of times. The labeling and construction for this can be done differently, but this is a labeling method based upon single rotations within each subgraph and will suffice. Note that when a subgraph is rotated n times counterclockwise every subgraph within that subgraph is also rotated n times counterclockwise.

Mapping 7.9 *A finite state function from a labeling based on single rotations within each subgraph to the base labeling is done using the following λ and δ functions, where $a, b, c \in \{0, \ldots, d-1\}$:*

$\lambda(a, q_b) = (a + b) mod\, d$, *and*

$\delta(a, q_b) = (c + b) mod\, d$, *where c is the number of rotations (counterclockwise) associated with the number a (this is the number of times that the ath subgraph will be rotated).*
The strings are read from left to right, and the starting state is q_0.

Theorem 7.10 *Mapping 7.9 is a finite state function mapping from the labeling based on single rotations within each subgraph to the base labeling.*

Proof: Let S be a string labelled with a method based upon single rotations. The first character is mapped to its subgraph number by the finite state function. We now assume that the first $n-1$ characters form the left of the string S are mapped to the proper base labelled characters. Consider the nth character from the left of the string S, and assume that there has been k previous rotations. This character (number) should have been rotated k times counterclockwise. Therefore if we add k to the nth character of S mod(d), we get the base labelled character. (Adding k to that character (n^{th} character) gives back the number that was originally there before being rotated, which was just the base labelled character.)That is because, that character started out as the base labelled character and was then rotated k times, (This idea should be clear from the description of a labeling based on single rotations within each subgraph). This is exactly what the finite state function does. Therefore, the finite state function from a labeling based on single rotations within each subgraph to the base labeling as defined above is a mapping from a labeling based on single rotations within each subgraph to the base labeling.

■

It is important to notice that you can choose any subgraph to rotate as many times as you would like, mod(d).

Corollary 7.11 *A labeling method based on single rotations within each subgraph has a finite state machine for codeword recognition.*

Proof: We have seen in corollary 7.5 that the base labeling has a finite state machine for codeword recognition. Theorem 7.10 showed that there is a finite state mapping from a labeling based on single rotations within each subgraph to the base labeling. Thus lemma 6.1 tells us that a labeling method based on a single rotations within each subgraph has a finite state machine for codeword recognition.

■

7.2 A Family Of Labelings With Finite State Characteristics

This section will introduce a family of labelings that have finite state machines for codeword recognition. From labelings based on single rotations within each subgraph we have d^d labelings that have finite state machines for codeword recognition. There are d choices on how to rotate each of the subgraphs, thus d^d labelings. Note that theorem 7.10 proved this fact. We can also combine rotations and reflections within a finite state way to create more choices of labelings. This will be done by first doing any of the d^d labelings based upon single rotations. Then applying a labeling based on a a single reflection to this labeling. Any of these labelings will have finite state characteristics as well.

Theorem 7.12 *Any labeling method based on single rotations and then a single reflection has a finite state machine for codeword recgonition.*

Proof: To show that this fact is true we will explain why there exists a composition of finite state functions from such a labeling to the base labeling. Consider a labeling based upon single rotations and then a single reflection. By applying the finite state function for mapping a single reflection labeling method to the base labeling method we will get back to the labeling method based upon single rotations. Then by applying the finite state function for mapping a labeling based upon single rotations to the base labeling we will get back to the base labeling. Therefore by the composition of two finite state functions we get to the base labeling. Since the base labeling possess a finite state machine for codeword recognition, lemma 6.1 tells us that any labeling method based on single rotations and then a single reflection will have a finite state machine for codeword recognition.

■

We have already stated that there are d^d labelings based upon single rotations. Now for each of these there are d ways to apply a single reflection. Therefore we see that there are d^{d+1} ways to have a labeling based upon single rotations and single reflections. The question now arises; How many of these labelings are distinct?

Theorem 7.13 *There are d^{d+1} distinct labelings based on single rotations and single reflections.*

Proof: Assume that there are two distinct labelings based upon single rotations and a single reflection that result in the same overall labeling method. Since these labelings are done

differently, they differ in the number of reflections and/or rotations. we will consider two cases.

Case 1: We assume that the two labelings differ in the number of rotations that they apply to at least one subgraph. If this is the case the K_d^2 graphs must differ for both of the labeling methods. Therefore they can not result in the same overall labeling method.

Case 2: We assume that the two labelings differ in the single reflection that is applied. If this is the case, the K_d^2 graphs must differ for both of the labeling methods. Therefore they can not result in the same overall labeling method.

Since this holds for both cases we conclude that there are d^{d+1} distinct labelings based on single rotations and single reflections.

■

The α-method labeling belongs to this family of labelings. It is a labeling based upon single rotations in each subgraph. This follows directly from how both labelings are defined. However, the C-R-E-L labeling in not a labeling based upon single rotations and single reflections.

Theorem 7.14 *The C-R-E-L labeling in not a labeling based upon single rotations and single reflections.*

Proof: Assume that the C-R-E-L labeling is a labeling based upon single rotations and single reflections. From the K_d^2 graph we see that the C-R-E-L labeling is actually a labeling based upon single rotations. No rotations in the zero subgraph, one rotation in the first subgraph, and two rotations in the second subgraph, (no single reflections occur). Upon examination of the K_d^3 graph we find that this rotational scheme has not continued to take place. Therefore the C-R-E-L labeling is not a labeling based upon single rotations and single reflections.

■

7.3 Single Permutation Labelings

This section will introduce another family of labelings that has a finite state machine for codeword recognition. This family of labelings is based upon the symmetric group on n letters.

Definition 7.15 *Let n be a positive integer, and consider the set, S_n, of all the permutations (bijective functions) from the set $n = \{1, 2, \ldots, n\}$ to itself. Under the operation of function composition, S_n is a group, called the **symmetric group on n letters**. Note that this is a finite group of order n!*

We will now describe the labeling based upon single permutations. For K_d^1 label the top vertex 0, and the other vertices with a distinct $i \in \{1, \ldots, d-1\}$. To label K_d^n first make d copies of labelled K_d^{n-1} and connect them using the **swap connection rule**. Then prefix an i to the left of each string in the C_i subgraph. Then for the second letter from the left of the string apply the permutation $\pi_{i,j}$, where i represents the subgraph number, and j specifies a certain permutation from S_d (symmetric group on d letters) where j is based upon the iteration number n. The j has to be based on something finite. For instance we can make j

based on numbers mod($d!$). (this is only one way, but seems like a good number since there are $d!$ permutations in S_d.)

To prove that this labeling has a finite state machine for codeword recognition we will use a finite state function to map it to the base labeling. To illustrate an example of what the $\pi_{i,j}$ permutations look like, we will show $\pi_{i,j}$ such that j is the iteration number taken mod($d!$). This is one of the most complicated cases, and any other method based upon using j differently can be gotten from modifying this case. For a labeling based on these $\pi_{i,j}$ permutations we have d $\pi_{i,j}$ permutations where,

$$\pi_{i,j} = \begin{cases} P_0, & \text{if} \quad j \equiv 0 \bmod(d!); \\ P_1, & \text{if} \quad j \equiv 1 \bmod(d!); \\ \vdots & \qquad\qquad \vdots \\ P_{d!-1}, & \text{if} \quad j \equiv (d!-1) \bmod(d!). \end{cases}$$

and P_0, ..., $P_{d!-1}$ are not necessarily unique permutations in S_d. The mapping is done using arbitrary $\pi_{i,j}$ permutations. The number of states will depend upon the $\pi_{i,j}$ permutations but will always be a specific finite number. To do this mapping in a finite state manner we need to know the length of a specific string modulos a specific integer k, where k is predetermined. This can be done using a finite state machine. Thus figuring out the length of a string and then apply the following mapping will be a finite state mapping.

Mapping 7.16 *The mapping from the labeling based upon single permutations to the base labeling is done by the finite state function consisting of the following* λ *and* δ *function, where* $a, i \in \{0, \ldots, d-1\}$, *and* j *is based upon the string length taken mod(k) for some integer k:*

$$\lambda(a,\, q_{i,j}) = q_{i,j}(a)$$

$$\delta(a,\, q_{i,j}) = q_{q_{i,j}(a),(j-1)(mod\, k)}.$$

The strings are read from left to right, and the state $q_{i,j}$ *is represented by the inverse permutation of* $\pi_{i,j}$ *in the sense that* $q_{i,j}(a)$ *is* $(\pi_{i,j})^{-1}(a)$. *The initial state is* q_0, *which represents the identity permutation.*

We will now show that this is a finite state mapping from this labeling based upon single permutations to the base labeling method.

Theorem 7.17 *The above mapping is a finite state mapping from the labeling based upon single permutations to the base labeling method.*

Proof: Consider a string of length 1. We see that the finite state mapping gives the proper base labelled character. We assume that a string of length $n-1$ is mapped to the proper base labelled string. Consider an arbitrary string, S, of length n labelled using the single permutation method. The only thing that will be different in this string as compared to the string of length $n-1$ (throwing out the first number on the left) is the first two characters. The finite state mapping will map the first (leftmost) character to itself. This tells us the subgraph number at the nth iteration. The character that is second from the left had the $\pi_{i,j}$

permutation applied to it, where i is the previous character and $j \equiv k (\bmod d!)$. Therefore the $(\pi_{i,j})^{-1}$ permutation applied to the second character gives us the proper base labelled second character. Therefore the finite state mapping maps S to the proper base labelled string, and the above described mapping is a finite state mapping from the labeling based upon single permutations to the base labeling method.

■

Corollary 7.18 *The labeling method based upon single permutations in each subgraph has a finite state machine for codeword recognition.*

Proof: We have seen in corollary 7.5 that the base labeling has a finite state machine for codeword recognition. Theorem 7.17 showed that there is a finite state mapping from a labeling based on a single permutation within each subgraph to the base labeling. Therefore lemma 6.1 tells us that a labeling method based on a single permutation within each subgraph has a finite state machine for codeword recognition.

■

This labeling method is much more powerful than the labeling methods based upon single reflections and the labeling method based upon single rotations in each subgraph. Both of these labeling methods are contained within the labeling method based upon single permutations within each subgraph. The question arises as to how many labelings can we create with finite state characteristics.

Theorem 7.19 *There are at least countably infinitely many labelings of K_d^n for any value d that have a finite state machine for codeword recognition.*

Proof: Theorem 7.18 tells us that any labeling of K_d^n based upon single permutations has a finite state machine for codeword recognition. We will show that there are at least countably infinitely many of these labelings. Consider the following $\pi_{i,j}$ permutations that we will apply to all d subgraphs at every iteration.

$$\pi_{i,j} = \begin{cases} P_0, & \text{if} \quad j \equiv 0 \bmod (k); \\ P_1, & \text{if} \quad j \not\equiv 0 \bmod (k). \end{cases}$$

Where P_0 and P_1 are specific permutations in the symmetric group S_d, and k is any integer. For any integer k we see that this produces a distinct labeling of K_d^n. Also, each of these labelings will have finite state machines for codeword recognition and error correction. Since there are countably infinitely many integers, we conclude that there are at least countably infinitely many labelings of K_d^n for any value d that have a finite state machine for codeword recognition.

■

8 Non-Finite State Labelings

Up until this point we have solely discussed labelings with finite state characteristics. This section will describe an infinite family of labelings without finite state properties for codeword recognition and error correction.

Theorem 8.1 *There exists labeling methods such that codeword recognition and error correction are not finite state.*

Proof: To prove this fact we will simply provide an example. For n prime construct K_d^n using the α-method labeling, and for n composite construct K_d^n using the base labeling method. Assume that this labeling method does have finite state machines for codeword recognition and error correction. Then somehow, these finite state machines would have to differentiate between all of the prime and composite numbers. However, finite state machines cannot distinguish between all the prime and composite numbers [6]. Therefore this labeling method cannot have these finite state characteristics. Thus we have shown the existence of a labeling method where codeword recognition and error correction are not finite state processes.

■

Theorem 8.2 *There exists infinitely many labeling methods such that codeword recognition and error correction are not finite state.*

Proof: From theorem 7.19 we know that there are countably infinitely many labelings with finite state properties. Let i denote such an arbitrary labeling. Then for n prime construct K_d^n using the base labeling, and for n composite construct K_d^n using the i labeling method. From theorem 8.1 we see that this is a labeling method that does not have finite state machines for error correction and codeword recognition. Since there are countably infinitely many labelings i, we now have countably infinitely many labelings with no finite state machines for error correction and codeword recognition.

■

9 Conclusion

We have presented a construction which will produce perfect one-error correcting code on any iterated complete graph K_d^n. We developed a method of labeling, CS method, that contains all of the labeling methods discussed by Alspaugh, Knight, and Meloney, [2], that is we have finite state mappings going from the CS method to each of these other methods. We introduce two families of labelings that have finite state machines for error correction and codeword recognition. One of the families, based upon single permutations, is countably infinite. This tells us that there are countably infinitely many labelings that have a finite state machine for codeword recognition. We proved the existence of labelings that do not have finite state machines for codeword recognition and error correction, and provide an infinite family of such labelings.

There are still some open questions that are left to be addressed. It is possible to show that all of the families of labelings that we have come up with have a finite state machine for error correction. The following lemma will be extremely useful:

Lemma 9.1 *Let A and B be distinct labeling methods such that the labeling method B has a finite state machine for error correction. If there are finite state functions from A to B and from B to A then A has a finite state machine for error correction.*

Also, one could look at all of the labeling methods with easy encoding and decoding. What labeling method produces the Towers of Hanoi labeling when $d = 3$? Why is the Towers of Hanoi labeling so special? Are there any other labeling methods that are based upon fun puzzles?

References

[1] Cull, P. and I. Nelson. *Perfect Codes, NP-Completeness, and Towers of Hanoi Graphs.* Bulletin of the ICA, Volume 26 (1999), pages 13-38.

[2] Alspaugh, S., N. Knight, and K. Meloney *Perfect One Error Correcting Codes on Iterated Complete Graphs.* Proceedings of the REU Program in Mathematics. NSF and Oregon State University. Corvallis, Oregon. August, 2001.

[3] Linz, P. *An Introduction to Formal Language and Automata.* Jones and Bartlett Publishers, 2001.

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

[5] Biggs, N. Perfect codes on graphs. *J. Combinatorial Theory(B)* 15:289-296, 1973.

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