PERFECT ONE-ERROR-CORRECTING CODES ON ITERATED COMPLETE GRAPHS: ENCODING AND DECODING FOR THE SF LABELING

PAMELA RUSSELL

ADVISOR: PAUL CULL
OREGON STATE UNIVERSITY

ABSTRACT. Birchall and Tedor proved that every iterated complete graph has a perfect one-error-correcting code (P1ECC) and showed how to construct it [2]. Kleven created the SF labeling method, a method for assigning strings to the vertices of iterated complete graphs which has several nice properties [5]. We use these results to create a "working" P1ECC on these graphs. That is, we present a method for encoding and decoding: an algorithm which gives a bijection between the set of messages which can be transmitted using the P1ECC on an iterated complete graph, and the set of codewords in the SF labeling of that graph. In the process, we introduce a technique which should be useful in creating encoding and decoding methods for any reasonable labeling of iterated complete graphs.

1. INTRODUCTION

Many factors can cause errors in the transmission of messages. It is for this reason that error-correcting codes have been developed. An error-correcting code is designed to recognize that the received message contains an error, then find the error and correct it. A biological example of an error-correcting code is written communication by humans. If an English speaker reads the word INFORMETION, he will automatically correct it to INFORMATION.

This paper is concerned only with digital codes. In particular, we look at one-error-correcting codes on graphs. In this context, words are represented by vertices in the graph, and each word that contains an error is adjacent to the corresponding word containing no errors. These codes come with algorithms for recognizing whether a word contains an error and correcting the error if it does. So in our example above, the vertex labeled INFORMETION would be adjacent to the vertex labeled INFORMATION, and we would have an algorithm which would locate the error in INFORMETION and correct the E to an A.

We look at codes on a particular family of graphs, iterated complete graphs. Much is known about codes on these graphs. In particular, every iterated complete graph has exactly one perfect one-error-correcting code up to isomorphism [2]. Kleven gave a method for assigning strings to the vertices of any odd-dimension iterated complete graph, called the SF labeling method, and also gave finite-state machines which recognize and correct errors in a given string [5].

Our contribution is to give an algorithm for encoding and decoding messages. That is, we give a bijection between the set of distinct messages which can be sent using a given iterated complete graph, and the set of codewords in the SF labeling of that graph.

Date: August 2004.

This work was done during the Summer 2004 REU program in Mathematics at Oregon State University.

2. Background: Iterated Complete Graphs and Error-Correcting Codes

2.1. Iterated Complete Graphs: Definitions.

Definition 2.1.1. *A* ***(simple) graph*** $G = (V,E)$ *consists of a finite set* V *(called* ***vertices****) and a set* E *(called* ***edges****). Elements of* E *are unordered pairs of elements of* V*. Two vertices* v_1 *and* v_2 *are* ***adjacent*** *(have an edge between them) if* $(v_1,v_2) \in E$*. The adjective "simple" indicates that any two vertices have at most one edge between them, and that no vertex is adjacent to itself.*

Definition 2.1.2. *The* ***degree*** *of a vertex* v *is the number of vertices which are adjacent to* v*.*

Definition 2.1.3. *The* ***complete graph*** *on* d *vertices, denoted* K_d*, is the graph such that all the vertices are pairwise adjacent.*

Figure 1 shows K_3, K_4, and K_5.

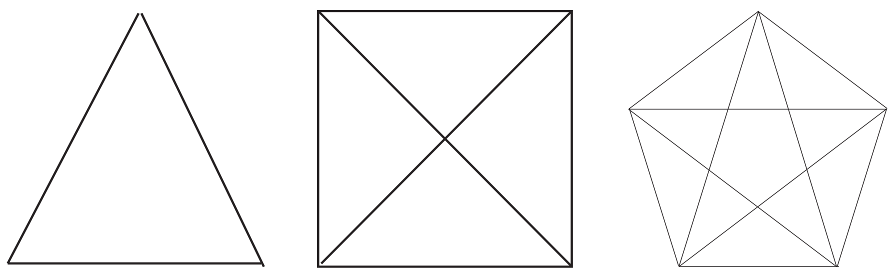

FIGURE 1. K_3, K_4, and K_5.

Definition 2.1.4. *The* ***second-order iterated complete graph****, denoted* K_d^2*, is constructed by making* d *copies of* K_d*, then connecting each pair of copies by one edge such that no vertex ends up with degree* $> d$*. The* ***order n iterated complete graph****, denoted* K_d^n*, is constructed by making* d *copies of* K_d^{n-1}*, then connecting each pair of copies by an edge between a corner vertex of one copy and a corner vertex of the other copy, such that no vertex ends up with degree* $> d$*. (Note: a* ***corner vertex*** *of* K_d^n *is a degree* $d-1$ *vertex.)*

Figure 2 shows K_5^1, K_5^2, and K_5^3.

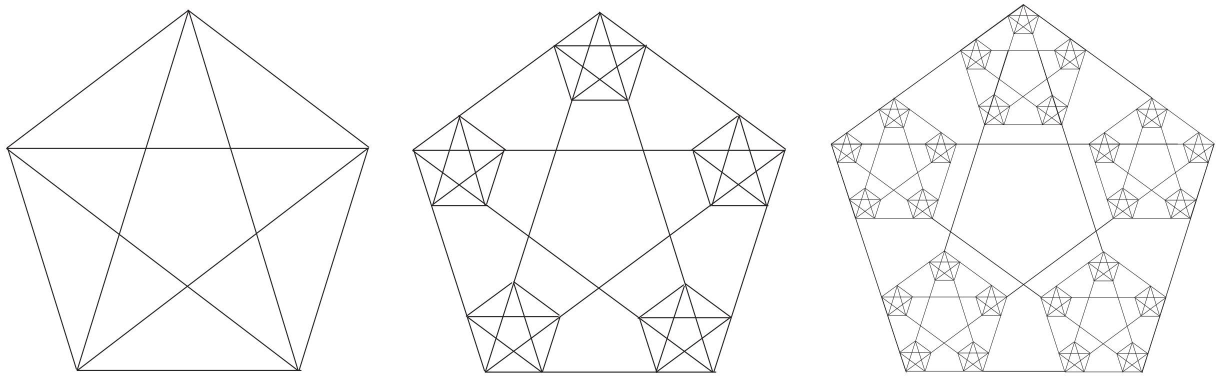

FIGURE 2. K_5^1, K_5^2, and K_5^3.

2.2. **Labelings.**

Definition 2.2.1. *A **labeling scheme** for K_d^n is a method of assigning strings of length n over $\{0,\ldots,d-1\}$ to the vertices of K_d^n such that this method gives a bijection between vertices and strings.*

Most labeling schemes have advantages and disadvantages. This paper deals only with the **SF labeling**, which has several desirable properties as we will see later [5]. Examples of other labeling schemes are the C-R-E-L labeling [2], the α-Method [1], the Multiples of $d+1$ code [1], and the C-S labeling method [4].

2.3. **Codes on Graphs; Perfect One-Error-Correcting Codes.**

Definition 2.3.1. *Let G be a graph and let V be the set of vertices of G. Then a **code** on G is a subset $C \subset V$. A **codevertex** is a vertex $c \in C$. A **noncodevertex** is a vertex $v \notin C$. If we have a labeling of G, then a **codeword** is the label of a codevertex. A **noncodeword** is the label of a noncodevertex.*

Definition 2.3.2. *A **perfect one-error-correcting code** (or **P1ECC**) on a graph G is a code such that:*

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

Most graphs have the property that no subset of their vertices is a P1ECC. In fact, Cull and Nelson showed that the problem of determining whether a given graph has a P1ECC is *NP*-complete [3]. However, Birchall and Tedor showed that every iterated complete graph has a P1ECC and that this P1ECC is unique up to isomorphism [2]. In Section 2.4, we give Alspaugh et al.'s construction of the P1ECC on K_d^n [1].

Figure 3 shows three examples of P1ECC's on iterated complete graphs.

Definition 2.3.3. *A **codeword recognizer** for a P1ECC is an algorithm for determining whether a given label is a codeword. An **error correction machine** sends a noncodeword to its corresponding codeword.*

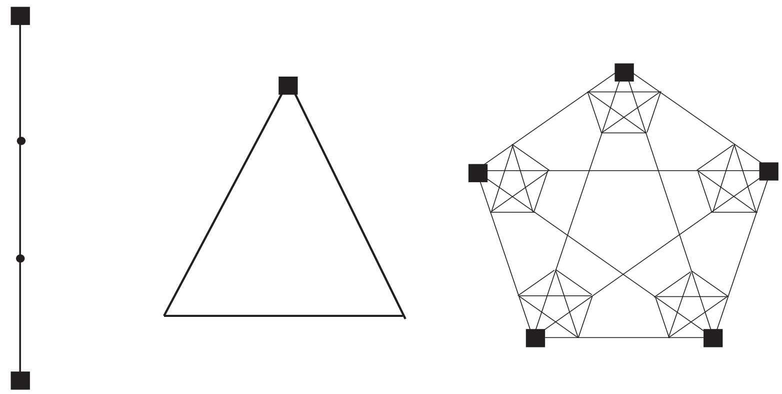

FIGURE 3. Perfect one-error-correcting codes on K_2^2, K_3^1, and K_5^2. Codevertices are represented by squares.

The adjective "perfect one-error-correcting" refers to the idea that the codewords are actual messages that one might wish to transmit. An error may be made in transmission, due to human error, interference, noise, etc. The set of vertices which are adjacent to a particular codevertex represents the set of errors that may be made when sending that message. The code is "*one*-error-correcting" because every noncodevertex is at a distance of one from a codevertex. If more errors are made, so that the actual transmitted message is no longer adjacent to the desired codevertex, then the message will be corrected to a different codeword.

Figure 4 shows an example where the two possible messages are LAND and SEA. An error-correction machine for this graph would correct the strings LQND, LADN, and LLLD to the codeword LAND. No other string would be corrected to LAND.

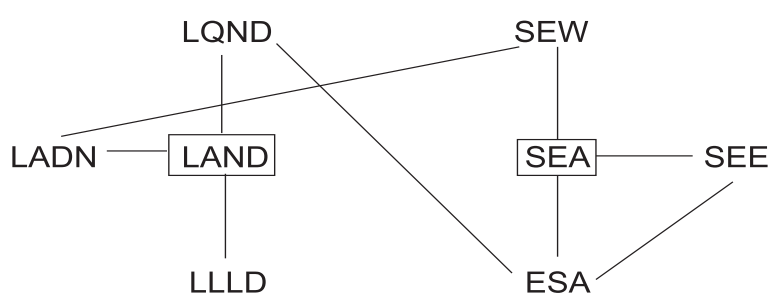

FIGURE 4. A P1ECC with two codewords (represented by rectangles).

Definition 2.3.4. *Let G be a graph. A labeling of G with the* **gray code property** *is a labeling such that any two adjacent vertices have labels which differ in exactly one position.*

One desirable property of the SF labeling is that it is a gray code. The labeling in Figure 4 does not have the gray code property. Figure 5 shows a labeling with the gray code property.

FIGURE 5. A labeling with the gray code property.

2.4. **The G-U Construction of a P1ECC on K_d^n.**

In this section, we give an iterative method for constructing the P1ECC on K_d^n. This method will be a cornerstone of our encoding and decoding scheme developed in sections 5, 6, and 7. This "G-U construction" is due to Birchall and Tedor [2]. However, we prefer the presentation given by Alspaugh et al. and we use it here [1].

The G-U construction uses two types of codes on K_d^n: G-codes and U-codes. The G-code is the desired P1ECC (Theorem 2.4.1). Let G_d^n denote K_d^n with the G-code and let U_d^n denote K_d^n with the U-code. G_d^n and U_d^n are constructed recursively as follows:

- To construct G_d^1, designate one vertex of K_d^1 as the *top vertex* and rotate it to the top position. Make this vertex a codevertex. Make the other $d-1$ vertices noncodevertices.
- To construct U_d^1, designate one vertex of K_d^1 as the top vertex and rotate it to the top position. Make all d vertices noncodevertices.

Figure 6 shows G_5^1 and U_5^1.

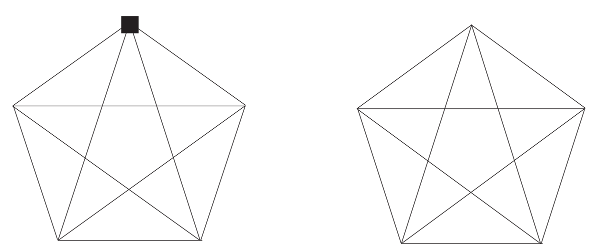

FIGURE 6. G_5^1 and U_5^1.

We now show how to construct G_d^n and U_d^n for arbitrary n:

To construct G_d^n when n is even:

(1) Make d copies of G_d^{n-1}.
(2) Connect each pair of copies by a vertex such that the top vertex of every copy remains unconnected.
(3) Designate the top vertex of some G_d^{n-1} as the top vertex of G_d^n.

To construct G_d^n when n is odd:

(1) Create one copy of G_d^{n-1} and $d-1$ copies of U_d^{n-1}.
(2) Connect the top vertices of the copies of U_d^{n-1} to distinct nontop corner vertices of G_d^{n-1}.
(3) Connect each pair of copies of U_d^{n-1} by one edge such that
  - This edge connects a nontop corner vertex in one copy to a nontop corner vertex in the other copy.
  - Exactly one nontop corner vertex of each U_d^{n-1} remains unconnected.
(4) Designate the top vertex of G_d^{n-1} as the top vertex of G_d^n.

To construct U_d^n when n is even:

(1) Make one copy of U_d^{n-1} and $d-1$ copies of G_d^{n-1}.
(2) Connect the top vertices of the copies of G_d^{n-1} to distinct nontop corner vertices of U_d^{n-1}.
(3) Connect each pair of copies of G_d^{n-1} by one edge such that
  - This edge connects a nontop corner vertex in one copy to a nontop corner vertex in the other copy.
  - Exactly one nontop corner vertex of each G_d^{n-1} remains unconnected.
(4) Designate the top vertex of U_d^{n-1} as the top vertex of U_d^n.

To construct U_d^n when n is odd:

(1) Make d copies of U_d^{n-1}.
(2) Connect each pair of copies by a vertex such that the top vertex of every copy remains unconnected.
(3) Designate the top vertex of some U_d^{n-1} as the top vertex of U_d^n.

Figure 7 shows G_5^2 and U_5^2. Figure 8 shows G_5^3 and U_5^3.

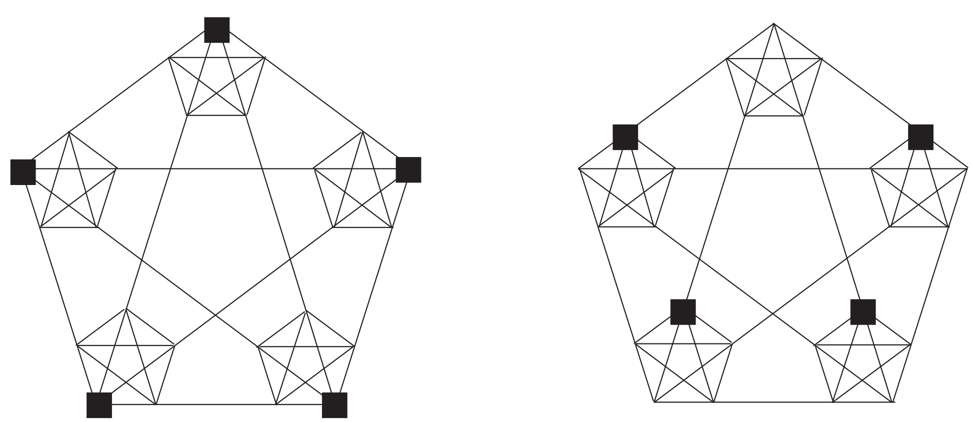

FIGURE 7. G_5^2 and U_5^2.

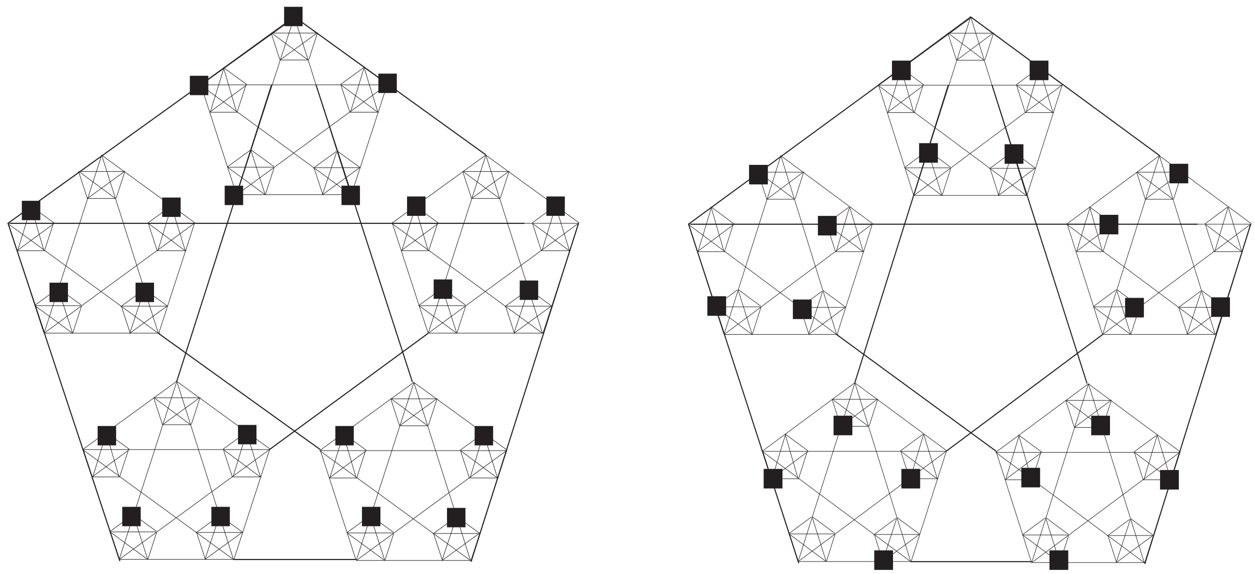

FIGURE 8. G_5^3 and U_5^3.

Theorem 2.4.1. *The G-code is the unique (up to isomorphism) P1ECC on K_d^n.*

Proof The proof is given by Birchall and Tedor [2] and is omitted here. ■

2.5. Encoding and Decoding.

Birchall and Tedor [2] showed that the number of codevertices c_n in a P1ECC on K_d^n is

$$c_n = \begin{cases} \frac{d^n+d}{d+1}, & n \text{ even} \\ \frac{d^n+1}{d+1}, & n \text{ odd.} \end{cases}$$

One can therefore use this code to transmit as many as c_n different messages. Label the possible messages by the first c_n natural numbers, i.e. the set $\{0,\ldots,c_n-1\}$.

Definition 2.5.1. *An* ***encoding and decoding scheme*** *for a particular labeling of K_d^n is a bijection between the set $\{0,\ldots,c_n-1\}$ and the set of codewords in the labeling.*

A useful encoding and decoding scheme works for all d and n. This way, depending on the number of messages one wishes to be able to encode, one can choose suitable d and n such that c_n is large enough. It is not necessary to generate the whole labeling and assign messages to codewords; one simply selects the message one wants to send and the encoding scheme returns the corresponding codeword. After the codeword is transmitted, the receiving party runs the codeword recognizer, the error corrector if necessary, and finally the decoder.

Some labelings have simple encoding and decoding schemes. For example, the SF labeling has a nearly trivial scheme when $d = 3$. However, this does not appear to be the case for the SF labeling with $d > 3$. In Sections 5 and 6, we lay out a technique that should be useful in finding encoding and decoding schemes for any reasonable labeling. In Section 7, we use this technique to give an encoding and decoding scheme for the SF labeling with arbitrary d and n.

3. The SF Labeling of K_d^n

In this section, we describe the construction of the SF labeling. (In Section 4, we show that there is no "similar" labeling of K_d^n when d is even.) We then give finite state machines for codeword recognition and error correction.[1]

3.1. **The SF Labeling.**

The SF labeling is only defined when d is odd. Let $d \geq 3$ be an odd number. The labeling of K_d^n is constructed recursively from the labeling of K_d^{n-1}.

Label K_d^1 as follows: the top vertex is labeled 0, then the remaining vertices are labeled $1, 2, \ldots, (d-1)$, going counterclockwise. Figure 9 shows the SF labeling of K_5^1.

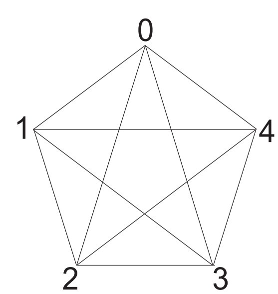

FIGURE 9. The SF labeling of K_5^1.

The SF labeling of K_d^n is constructed according to the following algorithm: Apply the permutation α to each digit in every label of K_d^{n-1}, where $\alpha(z) = \frac{d+1}{2} z \pmod d$. Now make d copies of $\alpha(K_d^{n-1})$. Rotate the k^{th} copy $\frac{2\pi k}{d}$ radians counterclockwise, then append k to each word in this copy. Finally, connect the d copies to form K_d^n. Figure 10 shows the SF labeling of K_7^2. Figure 11 shows the SF labeling of K_5^3.

[1] All results in Section 3 are due to Kleven [5].

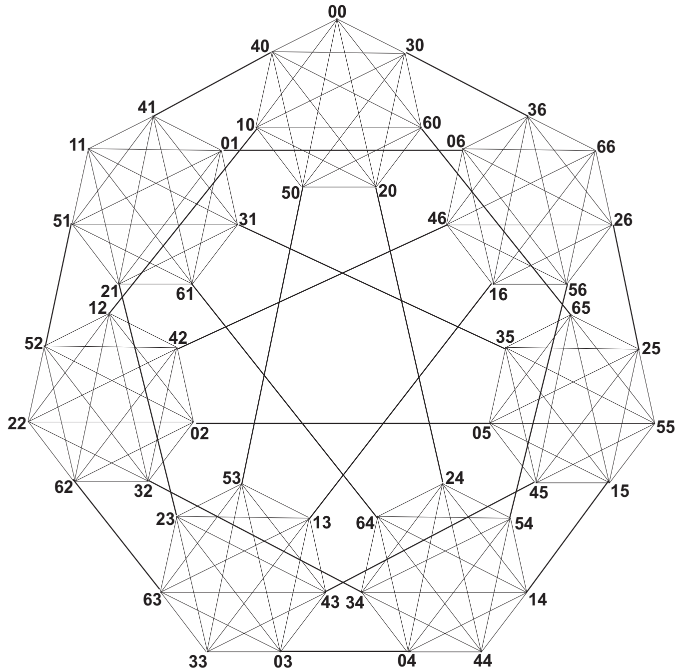

FIGURE 10. The SF labeling of K_7^2.

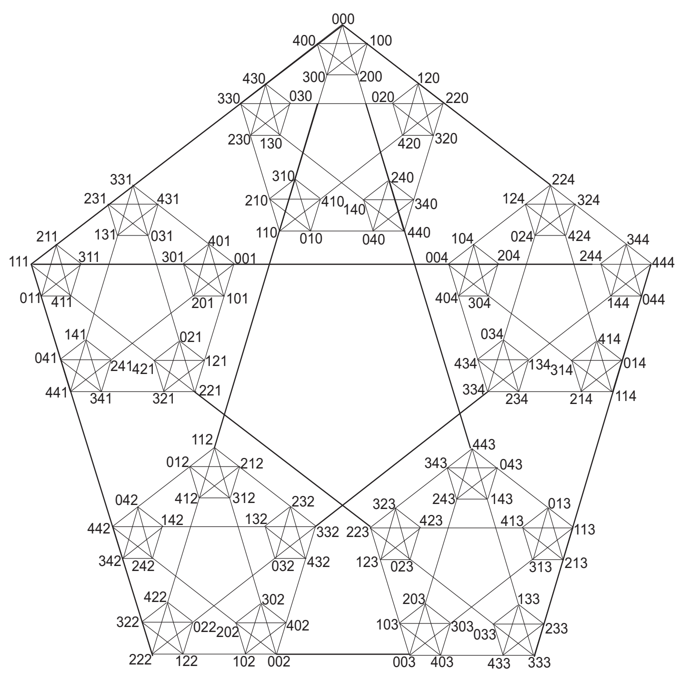

FIGURE 11. The SF labeling of K_5^3.

3.2. **Description of the Codewords.**

No explicit description of the codewords has been found. Kleven gives a recursive description. Let G_n denote the set of codewords in the SF labeling of K_d^n, and let U_n denote the set of SF labels of vertices in the U-code on K_d^n. Then,

When n is even,

$$G_n = G_{n-1} \circ 0 \cup T(G_{n-1}) \circ 1 \cup T^2(G_{n-1}) \circ 2 \cup \cdots \cup T^{d-1}(G_{n-1}) \circ (d-1)$$
$$U_n = U_{n-1} \circ 0 \cup \Gamma_1(G_{n-1}) \circ 1 \cup \Gamma_2(G_{n-1}) \circ 2 \cup \cdots \cup \Gamma_{d-1}(G_{n-1}) \circ (d-1)$$

And when n is odd,

$$G_n = G_{n-1} \circ 0 \cup \Gamma_1(U_{n-1}) \circ 1 \cup \Gamma_2(U_{n-1}) \circ 2 \cup \cdots \cup \Gamma_{d-1}(U_{n-1}) \circ (d-1)$$
$$U_n = U_{n-1} \circ 0 \cup T(U_{n-1}) \circ 1 \cup T^2(U_{n-1}) \circ 2 \cup \cdots \cup T^{d-1}(U_{n-1}) \circ (d-1)$$

where $G_{n-1} \circ 0$ means the set G_{n-1} with a zero appended to every string in the set, T replaces each character x by $x+1$ (mod d), T^m means T composed with itself m times, and Γ_m replaces each character x by $\alpha(T^m(x))$.

3.3. **Codeword Recognition for the SF Labeling.**

Kleven gives a $(2d+2)$-state machine for codeword recognition. The states are $\{\ E*S\ ,\ E*0\ ,\ E*1\ ,\ldots,\ E*(d-1)\ ,\ O*S\ ,\ O*0\ ,\ O*1\ ,\cdots,\ O*(d-1)\ \}$. The machine starts in state $E*S$ and reads strings from left to right. Strings of even length are accepted in the $E*S$ state and strings of odd length are accepted in the $O*0$ state. The function δ determines transitions among the states:

$$\delta(x*y,z) = (\delta_1(x,z)*\delta_2(y,z))$$

where δ_1 and δ_2 are given by:

$$\delta_1(E,z) = O$$
$$\delta_1(O,z) = E$$
$$\delta_2(x,z) = (2x-z) \text{ (mod } d)$$
$$\delta_2(x,x) = S$$
$$\delta_2(S,z) = z$$

Kleven proves that codeword recognition is performed correctly by this finite state machine.

3.4. **Error Correction for the SF Labeling.**

Kleven showed that the SF labeling is a gray code (see Definition 2.3.4). Therefore, error correction involves changing exactly one digit. The error correction algorithm consists of two small algorithms.

The first algorithm is a finite-state machine; it is used only when the error is in the first digit. It starts in the S state if n is even and the O state if n is odd. The S state is the only non-final state. The function δ determines transitions among the states:

$$\delta(x,z) = (\tfrac{x+z}{2}) \pmod d$$
$$\delta(x,x) = S$$
$$\delta(S,z) = z$$

The second algorithm is used when the error is not in the first digit. To correct the word $x_1 \cdots x_n$:

IF $x_2 \cdots x_n \notin U_{n-1}$

Correct $x_1 = q$, where q is the final state after running $x_n \cdots x_2$ through the first machine

ELSE

FOR $i = 2 \cdots n$

IF $x_i \neq x_1$

IF $i \neq n$

Correct $x_i = 2x_1 - x_i \pmod d$

ELSE

Correct $x_n = 0$

BREAK

END FOR

Kleven proves that error correction is performed correctly by these algorithms.

4. Generalized Towers of Hanoi Labelings

It is natural to ask whether K_d^n admits a labeling similar to the SF labeling when d is even. We will specify what we mean by "similar to the SF labeling." We will then show that in fact, K_d^n (d even) does not admit any such labeling.

4.1. **The Towers of Hanoi Labeling of K_3^n.**

The Towers of Hanoi is a popular puzzle. It consists of three pegs and several disks of different radii which fit on the pegs. A solution to the puzzle is a configuration where all the disks are on one tower (see Figure 12 for a picture). One is allowed to move one disk at a time, with the constraint that no larger disk may be placed on top of a smaller disk.

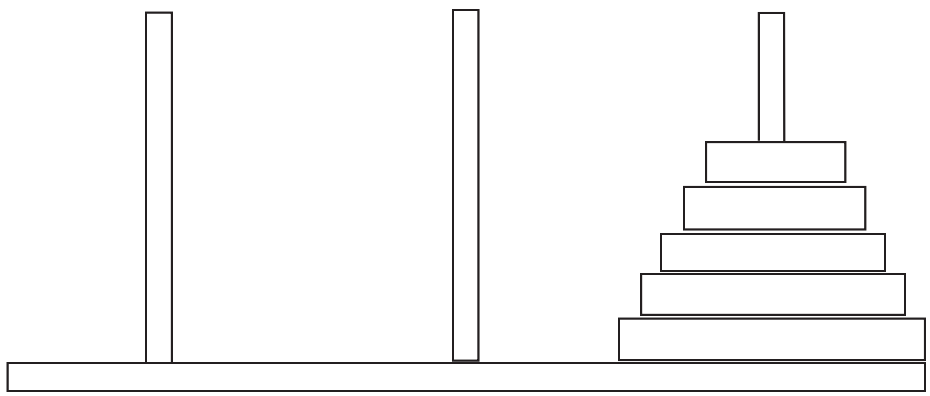

FIGURE 12. A solution to the Towers of Hanoi puzzle with five disks.

We can describe a configuration of the Towers of Hanoi puzzle by a string of length n over $\{0, 1, 2\}$, where n is the number of disks. The i^{th} digit represents the position of the i^{th} disk (first digit = smallest disk, n^{th} digit = largest disk). The position of a disk is 0 if the disk is on the leftmost tower, 1 if it is on the middle tower, and 2 if it is on the rightmost tower. See Figure 13 for an example. The configuration shown is 22201.

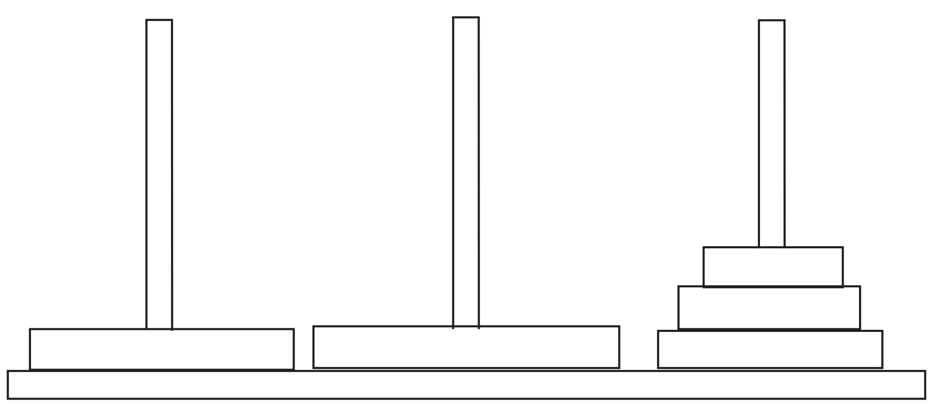

FIGURE 13. A configuration of Towers of Hanoi puzzle with five disks. The associated string is 22201.

One can construct a graph where each vertex is labeled with a configuration of the Towers of Hanoi puzzle and each edge represents a legal move in the puzzle. It turns out that this graph is actually K_3^n, and the labeling is known as the Towers of Hanoi labeling of K_3^n [3]. Furthermore, the resulting labeling is the SF labeling for $d = 3$ [5].

The rest of Section 4 examines the extent to which the SF labeling of K_d^n, $d > 3$, can be viewed as a generalization of the Towers of Hanoi labeling of K_3^n.

4.2. **Definitions.**

Definition 4.2.1. *The* ***Generalized Towers of Hanoi*** *puzzle on d towers and n disks, denoted GTH(d,n), is the puzzle with the following rules:*

(1) *Only one disk may be moved at a time.*
(2) *No larger disk may be placed on top of a smaller disk.*

The towers are labeled $0 \ldots d-1$. *A* ***configuration*** *of GTH(d,n) is a string of length n over* $\{0,\ldots,d-1\}$, *where the* i^{th} *digit gives the position (tower) of the* i^{th} *disk (first digit = smallest disk;* n^{th} *digit = largest disk).*

Definition 4.2.2. *Let G be any graph with at most* d^n *vertices. A* ***GTH(d,n) labeling*** *of G is a one-to-one map from the set of vertices of G to the set of configurations of GTH(d,n), such that each edge represents a legal move in GTH(d,n).*

Note: The SF labeling of K_d^n, with d odd, is a GTH(d,n) labeling. We prove this in Section 4.4.

Definition 4.2.3. *The* ***Maximal GTH(d,n) graph****, denoted* $M(d,n)$, *is the labeled graph on* d^n *vertices with edges corresponding to every legal move in GTH(d,n). Figure 14 shows* $M(4,2)$.

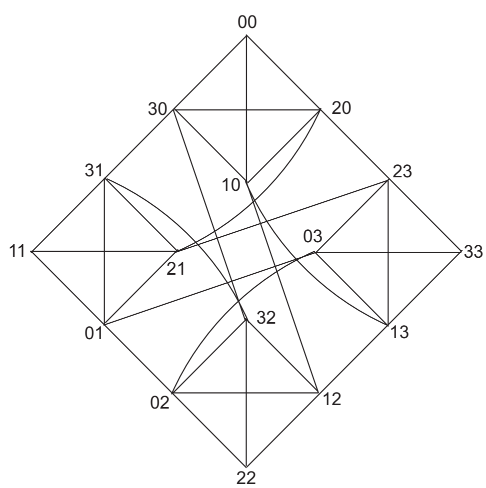

FIGURE 14. $M(4,2)$.

4.3. **Nonexistence of a GTH Labeling for d Even.**

We now show that there is no GTH(d,n) labeling of K_d^n when d is even.

Lemma 4.3.1. *If there is no GTH(d,$n-1$) labeling of K_d^{n-1}, then there is no GTH(d,n) labeling of K_d^n.*

Proof We prove the contrapositive.

Suppose we have a GTH(d,n) labeling of K_d^n. Recall that K_d^n consists of d copies of K_d^{n-1}. Call them $C_0, \ldots, C_{d-1}$. Delete all the vertices in $C_1, \ldots, C_{d-1}$ so that only C_0 remains. Now delete the n^{th} character in each vertex label in C_0 (this corresponds to removing the largest disk). The result is a GTH(d,$n-1$) labeling of K_d^{n-1}. ■

Theorem 4.3.2. *For d even, there is no GTH(d,n) labeling of K_d^n.*

Note: We are assuming that $n > 1$ since there is always a GTH(d,1) labeling of K_d^1. Also, we assume that $d > 2$ since a GTH puzzle on two towers is not interesting.

Proof We prove the theorem for $n = 2$ (two disks). The general result follows by induction, using Lemma 4.3.1.

Let $d > 2$ be even. We first distinguish between two types of edges in $M(d,2)$:

- **Type S Edges**: edges which correspond to moving the small disk
- **Type L Edges**: edges which correspond to moving the large disk

We try to construct a GTH(d,2) labeling of K_d^2. In other words, we must delete some of the edges in $M(d,2)$ so that the resulting graph is K_d^2. We will see that this is impossible.

Note that $M(d,2)$ contains exactly d copies of K_d^1, corresponding to the d possible positions of the large disk, from which the small disk may be moved to any tower. These d copies of K_d^1 will have to be the d copies of K_d^1 in K_d^2. Call them $C_0 \ldots C_{d-1}$ and keep their labelings. (Note that all the edges within C_i are Type S edges, and so far all the vertices in C_i have degree $d-1$.)

Now all that is left is to connect each C_i and C_j, $i \neq j$, by a Type L edge. This will require a total of $\binom{d}{2}$ Type L edges. If we look at only the Type L edges in $M(d,2)$, we see that they take the form of d distinct complete graphs on $d-1$ vertices. (Figure 15 illustrates this for $M(4,2)$.) This is because there are d ways to fix the position of the small disk, and then the large disk may be moved freely among the remaining $d-1$ towers. Call these complete graphs $D_0, \ldots, D_{d-1}$.

Recall that we are trying to connect each C_i and C_j by a Type L edge, so we need to use $\binom{d}{2}$ of the Type L edges which are found in the D_i's. This forces us to use at least $\frac{\binom{d}{2}}{d} = \frac{d-1}{2}$ edges from some D_i. Since d is even, this means that we must use at least $\frac{d}{2}$ edges from some D_i. But D_i has only $d-1$ vertices, so some vertex must belong to two of the $\frac{d}{2}$ chosen edges. Thus, this vertex will have degree at least $d+1$ in our constructed graph, which is impossible in K_d^2. ■

Note: The reason we can find a GTH(d,n) labeling of K_d^n when d is odd, is that $\frac{\binom{d}{2}}{2} = \frac{d-1}{2}$ is an integer which is equal to half the number of vertices in D_i, so no vertex need be used twice.

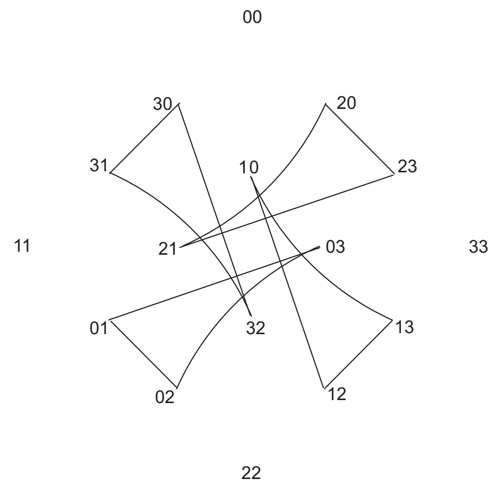

FIGURE 15. The type L edges in $M(4,2)$ form four triangles.

4.4. The SF Labeling is a GTH Labeling.

In this section we prove that the SF labeling of K_d^n is a GTH(d,n) labeling. We will need two lemmas.

Lemma 4.4.1. *For all $i \in \{0,\ldots,d-1\}$, we have $\frac{d+1}{2}i \equiv i - \frac{d+1}{2}i \pmod d$.*

Proof We derive this from a tautology:

$$\begin{aligned}
&1 \equiv 1 \pmod d\\
&\Longrightarrow d+1 \equiv 1-d \pmod d\\
&\Longrightarrow \tfrac{d+1}{2} \equiv \tfrac{1-d}{2} \pmod d\\
&\Longrightarrow \tfrac{d+1}{2} \equiv 1-\tfrac{d+1}{2} \pmod d\\
&\Longrightarrow \tfrac{d+1}{2}i \equiv i-\tfrac{d+1}{2}i \pmod d. \quad \blacksquare
\end{aligned}$$

Lemma 4.4.2. *The SF labeling assigns the strings $\underbrace{00\cdots0}_{n-times}$, $11\cdots1$, ..., $(d-1)(d-1)\cdots(d-1)$ to the corner vertices of K_d^n in order, starting with the top vertex and going counterclockwise.*

Proof We know by definition that the lemma is true for $n=1$. We prove that it is also true for $n=2$. This is sufficient to obtain the lemma for all n by induction, since there is only one digit of information contained in the word $ii\cdots i$.

Notation: We say that a vertex is in "position i" of K_d^1 if it is the i^{th} vertex, starting with zero at the top and counting counterclockwise. We say that K_d^2 contains d copies of K_d^1. We call them C_0, ..., C_{d-1}, starting with C_0 at the top and counting counterclockwise. So the j^{th} corner vertex of K_d^2 is the vertex which is in position j of C_j.

We want to show that the j^{th} corner vertex of K_d^2 is labeled jj. We go through the construction of K_d^2 to show that this is in fact true. To do this, we begin with the i^{th} vertex of K_d^1 and follow it through the construction.

The i^{th} vertex of K_d^1 is labeled i. The first step in the construction of the SF labeling relabels this vertex $\frac{d+1}{2}i \pmod d$. In the next step, we create d copies of the relabeled K_d^1. In the third step, we rotate the k^{th}

copy, C_k, by k positions clockwise. This means that when k is equal to $\frac{d+1}{2}i \pmod{d}$, the vertex labeled k (or equivalently, the vertex labeled $\frac{d+1}{2}i \pmod{d}$) is in position $i-k$. But by Lemma 4.4.1, we have that $k \equiv i-k$. So the vertex labeled k is in the k^{th} position of C_k (this is the k^{th} corner vertex of K_d^2.) In the fourth step, we append k to each vertex in C_k, so that kk is the k^{th} corner vertex of K_d^2. ■

We can now show that the SF labeling is a GTH labeling.

Theorem 4.4.3. *The SF labeling of K_d^n is a GTH(d,n) labeling for all $d \geq 3$.*

Proof We fix d and prove the theorem by induction on n.

The result is obvious for $n = 1$. Now assume that the SF labeling of K_d^{n-1} is a GTH(d,$n-1$) labeling. We examine the construction of K_d^n.

Note that after the first step in the construction (permuting the digits in the labeling of K_d^{n-1}), the resulting labeling is still a GTH(d,$n-1$) labeling. This is because permuting the digits in a configuration of the GTH puzzle is analogous to gluing all the disks to the towers they are currently on, then shuffling the towers around, disks and all. Legal moves are sent to legal moves.

So we need only worry about the edges that will be introduced between distinct copies of K_d^{n-1}. Call these copies D_0, ..., D_{d-1}.

Now, the clockwise rotation scheme guarantees that the edge between D_i and D_j ($i \neq j$) will connect two vertices with identical labels. Furthermore, these two vertices are corner vertices of D_i and D_j respectively, but they are not corner vertices of K_d^2, so by Lemma 4.4.2, they are labeled $aa\cdots a$ for some $a \neq i \neq j$.

The final step in the construction appends i to one vertex and j to the other, leaving an edge between $aa\cdots ai$ and $aa\cdots aj$. This represents a legal move since $a \neq i \neq j$. ■

5. An Indexing System for the Codevertices in K_d^n

In this section we give a method for representing a codevertex in K_d^n by an $(n-1)$-tuple over $\{0,\ldots,d-1\}$. The advantage of this technique is that the $(n-1)$-tuple contains explicit information, in a simple form, about the position of the codevertex inside K_d^n. This representation will later serve as an intermediate step in our encoding and decoding scheme.

5.1. **A Scheme for Representing Codevertices by Vectors.**

Let C_d^n be a P1ECC on K_d^n, let c_n be the number of codevertices in C_d^n, and pick $v \in C_d^n$. We give a recursive algorithm for assigning an $(n-1)$-tuple $(w_1, w_2, \ldots, w_{n-1})$ to v.

Algorithm. First of all, K_d^n contains d copies of K_d^{n-1}. Label them $0,\dots,d-1$ with 0 at the top. If v is contained in copy i, then let $w_1 = i$.

This K_d^{n-1} contains d copies of K_d^{n-2}. Label them $0,\dots,d-1$ with 0 being the top copy, where "top" is taken in the sense of the G-U construction. If v is contained in copy j, then let $w_2 = j$.

In general, suppose w_k ($k < n-2$) is given, so that v is contained in the $w_k{}^{\text{th}}$ copy of K_d^{n-k} inside a copy of K_d^{n-k+1}. If v is contained in the l^{th} copy of K_d^{n-k-1} inside this K_d^{n-k} (where the 0^{th} copy is the top copy, with "top" taken in the sense of the G-U construction), then let $w_{k+1} = l$.

Suppose w_{n-2} is given. There is a small change at this step. If v is contained in the p^{th} copy of G_d^1 inside a copy of K_d^2, then let $w_{n-1} = p$. Note that if this copy of K_d^2 is a U_d^2, then the p^{th} copy of G_d^1 is the $(p+1)^{\text{st}}$ K_d^1 subgraph.

Note: No two distinct codevertices have the same corresponding $(n-1)$-tuple since a copy of K_d^1 contains at most one codevertex.

Definition 5.1.1. *We say that an $(n-1)$-tuple η over $\{0,\dots,d-1\}$ **represents** a codevertex in C_d^n if there is some $v \in C_d^n$ whose corresponding $(n-1)$-tuple is η.*

5.2. A Map Between Natural Numbers and Codevertices.

Let Γ denote the set of $(n-1)$-tuples over $\{0,\dots,d-1\}$. We define a function

$$\Phi : \{0,1,\dots,d^{n-1}-1\} \longrightarrow \Gamma.$$

by

$$m \longmapsto (m \text{ in base } d, \text{ but written backwards}).$$

We give an explicit algorithm for Φ below. (Φ will later be pared down to a bijection between the set $\{0,\dots,c_n-1\}$ and the set of $(n-1)$-tuples which represent codewords in K_d^n. This bijection will provide a big piece of our encoding and decoding scheme.)

Let $m \in \{0,1,\dots,d^{n-1}-1\}$. The $(n-1)$-tuple

$$\Phi(m) = (v_1, v_2, \dots, v_{n-1})$$

is produced according to the following algorithm:

ALGORITHM

$t_0 = m$
$v_1 = t_0 \pmod d$
$k = 1$

WHILE $k < n-1$

$t_k = \frac{t_{k-1}-v_k}{d}$

$v_{k+1} = t_k \pmod d$

$k = k+1$

END WHILE

Notation. Let M denote the set $\{0,1,\ldots,c_n-1\}$. Let $C \subset \Gamma$ denote the set of $(n-1)$-tuples over $\{0,\ldots,d-1\}$ which represent codevertices in K_d^n.

Now that Φ is defined, we will proceed by the following steps:

(1) We first show that Φ is a bijection (Proposition 5.3.3).
(2) Next we show that if $m \in M$, then $\Phi(m) \in C$ (Lemma 5.3.6).
(3) These results give the theorem that $\Phi|_M$ is a bijection between M and C (Theorem 5.3.7).
(4) Finally, we give an explicit expression for the inverse map $\Phi^{-1} : C \longrightarrow M$ (Proposition 5.3.8).

5.3. **Properties of Φ.**

Our first goal here is to show that Φ is a bijection. This requires two lemmas.

Lemma 5.3.1. *Let $0 \le m_1 \le m_2 \le d^{n-1}-1$. Then $\Phi(m_2-m_1) = \Phi(m_2)-\Phi(m_1)$ (where subtraction is componentwise).*

Proof First of all, (m_2-m_1) (mod d) $= m_2$ (mod d) $-m_1$ (mod d).
$\Longrightarrow v_1(m_2-m_1) = v_1(m_2)-v_1(m_1)$.

Furthermore,

$$t_1(m_2-m_1) = \tfrac{m_2-m_1-v_1(m_2-m_1)}{d}$$

$$= \tfrac{m_2-m_1-v_1(m_2)+v_1(m_1)}{d} \quad \text{(by the above)}$$

$$= \tfrac{m_2-v_1(m_2)}{d} - \tfrac{m_1-v_1(m_1)}{d}$$

$$= t_1(m_2)-t_1(m_1).$$

The rest of the algorithm is completely determined by v_1 and t_1, so the lemma is proved. ■

Lemma 5.3.2. *Let $m \in \{0,\ldots,d^{n-1}-1\}$ and $\Phi(m) = (v_1,v_2,\ldots,v_{n-1})$, and suppose that $v_1 = v_2 = \ldots = v_k = 0$ for some $k \le n-1$. Then $m \equiv 0$ (mod d^k).*

Proof We first show that for all $0 \le j \le k-1$, we have $t_j = \frac{m}{d^j}$. This is clearly true for $j=0$ since $t_0 = m$. Now suppose $t_l = \frac{m}{d^l}$ for all $0 \le l \le j$. Then $t_{l+1} = \frac{t_l - v_{l+1}}{d}$. But $v_{l+1} = 0$ since $j \le k-1$, so $t_{l+1} = \frac{t_l}{d} = \frac{m}{d^{l+1}}$.

We can now prove the lemma. Suppose that $v_1 = \ldots = v_k = 0$. Then by the above, we have $t_{k-1} = \frac{m}{d^{k-1}}$. We also have $v_k = 0$, i.e., $t_k \equiv 0$ (mod d). So $\frac{m}{d^{k-1}} \equiv 0$ (mod d). Therefore m is divisible by d^k. ■

Proposition 5.3.3. *Φ is a bijection between $\{0,1,\ldots,d^{n-1}-1\}$ and Γ.*

Proof The preimage set and the target set have the same cardinality, so it is sufficient to show that Φ is one-to-one.

Suppose we have $m_1 \leq m_2$ such that $\Phi(m_1) = \Phi(m_2)$.
$\Longleftrightarrow \Phi(m_2) - \Phi(m_1) = (0,0,\ldots,0)$
$\Longrightarrow \Phi(m_2 - m_1) = (0,0,\ldots,0)$ by Lemma 5.3.1.
$\Longrightarrow m_2 - m_1$ is divisible by d^{n-1}, by Lemma 5.3.2.
$\Longrightarrow m_2 - m_1$ must be zero since $m_2 - m_1$ is nonnegative and the next multiple of d^{n-1} is not in $\{0,\ldots,d^{n-1}-1\}$.
$\Longrightarrow m_1 = m_2$. ∎

We now use Proposition 5.3.3 to show that by restricting the domain of Φ to M, we obtain a bijection between M and C. This uses three lemmas.

Lemma 5.3.4.

$$\Phi(c_n - 1) = \begin{cases} (\underbrace{0\ ,\ d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1}_{n-1}), & n \text{ odd} \\ (\underbrace{d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1}_{n-1}), & n \text{ even.} \end{cases}$$

Proof First note that

$$c_{n-1} = \begin{cases} \frac{d^n-1}{d+1} = d^{n-1} - d^{n-2} + d^{n-3} - \ldots + d - 1, & n \text{ even} \\ \frac{d^n-d}{d+1} = d^{n-1} - d^{n-2} + d^{n-3} - \ldots + d^2 - d, & n \text{ odd.} \end{cases}$$

When we write $c_n - 1$ in this form, the result is clear by inspection. ∎

Lemma 5.3.5. *The $(n-1)$-tuple in C which maximizes Φ^{-1} is*

$$\begin{cases} (0\ ,\ d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1), & n \text{ odd} \\ (d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1), & n \text{ even.} \end{cases}$$

Proof We prove the result by induction.

When $n = 2$, we have $c_2 = d$. So $M = \{0,\ldots,d-1\}$ and $\Phi(m) = (m)$. Therefore, $(d-1) \in C$ is the codevertex which maximizes Φ^{-1}. This $(n-1)$-tuple is of the desired form.

Now suppose the lemma is true in K_d^q.

Case 1: q is odd.

Then we are given that the codevertex in K_d^q which maximizes Φ^{-1} is $(\underbrace{0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1}_{q-1})$.

Since q is even, k_d^{q+1} consists of d copies of G_d^q, so there are d codevertieces in K_d^{q+1} which are of the form $(\underbrace{i\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1}_{q})$, $i = 0\ldots d-1$. So the codevertex in K_d^{q+1} which maximizes Φ^{-1} is $(d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1)$.

Case 2: q is even.
Then we are given that the codevertex in K_d^q which maximizes Φ^{-1} is $(\underbrace{d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1}_{q-1})$.

Now, K_d^{q+1} consists of one copy of G_d^q and $d-1$ copies of U_d^q. We look for i such that $(\underbrace{i, d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1}_{q})$ maximizes Φ^{-1} over the codevertices in K_d^{q+1}. But by Lemma 5.3.4, $\Phi^{-1}((\underbrace{d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1}_{q-1})) = c_q - 1$. So $(\ i\ ,\ d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1)$ cannot be in a K_d^q subgraph with fewer than c_q codevertices. Since U_d^q has fewer than c_q codevertices, $(\ i\ ,\ d-1\ ,\ 0\ ,\ d-1\ ,\ldots,\ 0\ ,\ d-1)$ must be in the copy of G_d^q, which is the top subgraph of K_d^{q+1}, so we are forced to pick $i=0$, and the codevertex in K_d^{q+1} which maximizes Φ^{-1} is of the desired form. ■

Lemma 5.3.6. *If $m \in M$, then $\Phi(m) \in C$.*

Proof Since Φ is a bijection, we prove the equivalent statement: if $(w_1, w_2, \ldots, w_{n-1}) \in C$, then $\Phi^{-1}((w_1, w_2, \ldots, w_{n-1})) \in M$. Fix $(w_1, w_2, \ldots, w_{n-1}) \in C$.

Case 1: $w_{n-1} \neq d-1$.
Then $w_1 \cdots w_{n-2}$ can be anything and this still represents a codevertex, so the result holds automatically.

Case 2: $w_{n-1} = d-1$.
Lemmas 5.3.4 and 5.3.5 tell us that the $(n-1)$-tuple which maximizes Φ^{-1} is exactly $\Phi(c_n-1)$. So there is no $c \in C$ with $\Phi^{-1}(c) \notin M$. ■

Theorem 5.3.7. *The restricted map $\Phi : M \longrightarrow C$ is a bijection.*

Proof Lemma 5.3.6 tells us that $\Phi(M) \subset C$. Furthermore, by Proposition 5.3.3, Φ must be a bijection between M and $\Phi(M)$. Therefore, since M and C have the same cardinality, $\Phi(M) = C$ and $\Phi : M \longrightarrow C$ is a bijection. ■

Finally, we give an explicit expression for Φ^{-1}.

Proposition 5.3.8. *Let $(w_1, w_2, \ldots, w_{n-1}) \in C$. Then*

$$\Phi^{-1}((w_1, w_2, \ldots, w_{n-1})) = d\ (\ d\ (\ \underbrace{\ldots}_{n-7 \text{ times}}\ (\ d\ (\ d\ (\ d \cdot w_{n-1} + w_{n-2}) + w_{n-3}) + w_{n-4}) + \underbrace{\ldots}_{n-7}) + w_2) + w_1.$$

(Call this number t_0.)

Proof By Proposition 5.3.3, $\Phi^{-1}((w_1, w_2, \ldots, w_{n-1}))$ exists. So it is sufficient to compute $\Phi(t_0) = (v_1, v_2, \ldots, v_{n-1})$ and see that $v_i = w_i$ for all $1 \le i \le n-1$. First note that $t_0 \equiv w_1 \pmod d$, so $v_1 = w_1$. So we have

$$t_1 = \frac{t_0 - w_1}{d} = d\ (\ d\ (\ \underbrace{\ldots}_{n-8 \text{ times}}\ (\ d\ (\ d\ (\ d \cdot w_{n-1} + w_{n-2}) + w_{n-3}) + w_{n-4}) + \underbrace{\ldots}_{n-8}) + w_3) + w_2$$

Now let $1 < k < n-2$ and suppose

$$t_{k-1} = d\ (\ d\ (\ \underbrace{\ldots}_{n-k-6 \text{ times}}\ (\ d\ (\ d\ (\ d \cdot w_{n-1} + w_{n-2}) + w_{n-3}) + w_{n-4}) + \underbrace{\ldots}_{n-k-6}) + w_{k+1}) + w_k$$

Then

$$t_k = \frac{t_{k-1} - w_k}{d} = d\ (\ d\ (\ \underbrace{\cdots}_{n-k-7\ times}\ (\ d\ (\ d\ (\ d \cdot w_{n-1} + w_{n-2}) + w_{n-3}) + w_{n-4}) + \underbrace{\cdots}_{n-k-7}) + w_{k+2}) + w_{k+1}$$

and $v_{k+1} = t_k \pmod{d} = w_{k+1}$. ■

6. Using Φ to Describe the Position of the m^{th} Codevertex in K_d^n

In this section we give an algorithm which uses $\Phi(m)$ to describe explicitely the position of the corresponding codevertex in K_d^n. We will arrive at a bijection between $\{0,\dots,c_n-1\}$ and the set of positions of codevertices. This is an extremely useful technique. In particular, it provides the basis for our encoding and decoding scheme for the SF labeling. Perhaps more importantly, though, it could be used to create an encoding and decoding scheme for any "reasonable" labeling of K_d^n.

6.1. Describing the Position of any Vertex in K_d^n: "Right-Side-Up" Coordinates.

Since there are d^n vertices in K_d^n, we can describe the position of each vertex by an n-tuple $(u_1, u_2, \dots, u_n)$ over $\{0,\dots,d-1\}$. Obviously, there are numerous ways to do this. We introduce a system called **Right-Side-Up** (RSU) coordinates.

In RSU coordinates, every K_d^m subgraph of K_d^n is viewed as being oriented the same way as K_d^n. If our vertex is contained in the i^{th} K_d^m subgraph of a K_d^{m+1} subgraph (where 0 points the same way as the top vertex of K_d^n and we count counterclockwise), then we simply let $u_{n-m} = i$. (Note: the vertex itself is a K_d^0 subgraph.)

As an example, Figure 16 shows the RSU coordinates of each vertex in K_5^2.

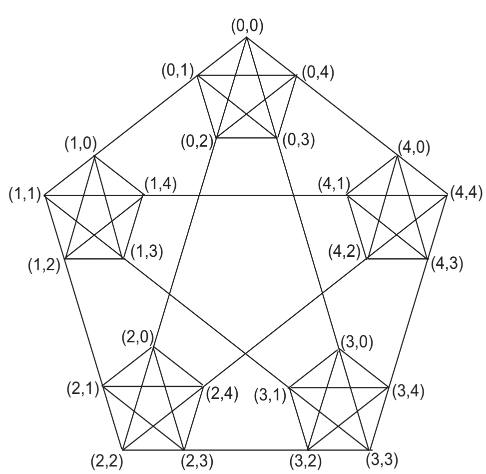

Figure 16. RSU coordinates on K_5^2.

In Section 6.2 we give the forward algorithm which takes in $\Phi(m)$ and returns the RSU coordinates of the m^{th} codeword. Section 6.3 contains a proof of the algorithm. In Section 6.4, we give the inverse algorithm, and in Section 6.5 we prove it. These steps give us the complete bijection between $\{0,\ldots,c_n-1\}$ and the RSU coordinates of codevertices.

6.2. **Forward Algorithm.**

We break the forward algorithm into two parts. The first part uses $\Phi(m)$ to produce the vector $S = (s_1,\ldots,s_n)$, whose i^{th} component is either $(G,\ n-i+1)$ or $(U,\ n-i+1)$, depending on whether the codevertex described by $\Phi(m)$ is contained in a G_d^{n-i+1} or a U_d^{n-i+1}. In the process, we also translate $\Phi(m)$ into what we refer to as the **Relative** coordinates (defined later in Definition 6.3.2) of the codevertex. The second part of the algorithm translates Relative coordinates into RSU coordinates.

FORWARD ALGORITHM: PART 1 (Produces the vector S. Figure 17 is a visual representation of this part of the algorithm.)

$(v_1,v_2,\ldots,v_{n-1}) = \Phi(m)$
$k = 1$
$s_1 = (G,n)$

WHILE $k < n$

  IF $s_k = (G,\ n-k+1)$

    IF $n-k+1$ is even

      $s_{k+1} = (G,\ n-k)$

    ELSE

      IF $v_k = 0$

        $s_{k+1} = (G,\ n-k)$

      ELSE

        $s_{k+1} = (U,\ n-k)$

  ELSE (i.e. if $s_k = (U,\ n-k+1)$)

    IF $n-k+1$ is even

      $v_k = v_k + 1$

      IF $v_k = 0$

        $s_{k+1} = (U,\ n-k)$

      ELSE

        $s_{k+1} = (G,\ n-k)$

    ELSE

      $s_{k+1} = (U,\ n-k)$

$k = k+1$

END WHILE

RETURN $S = (s_1, \ldots, s_n)$

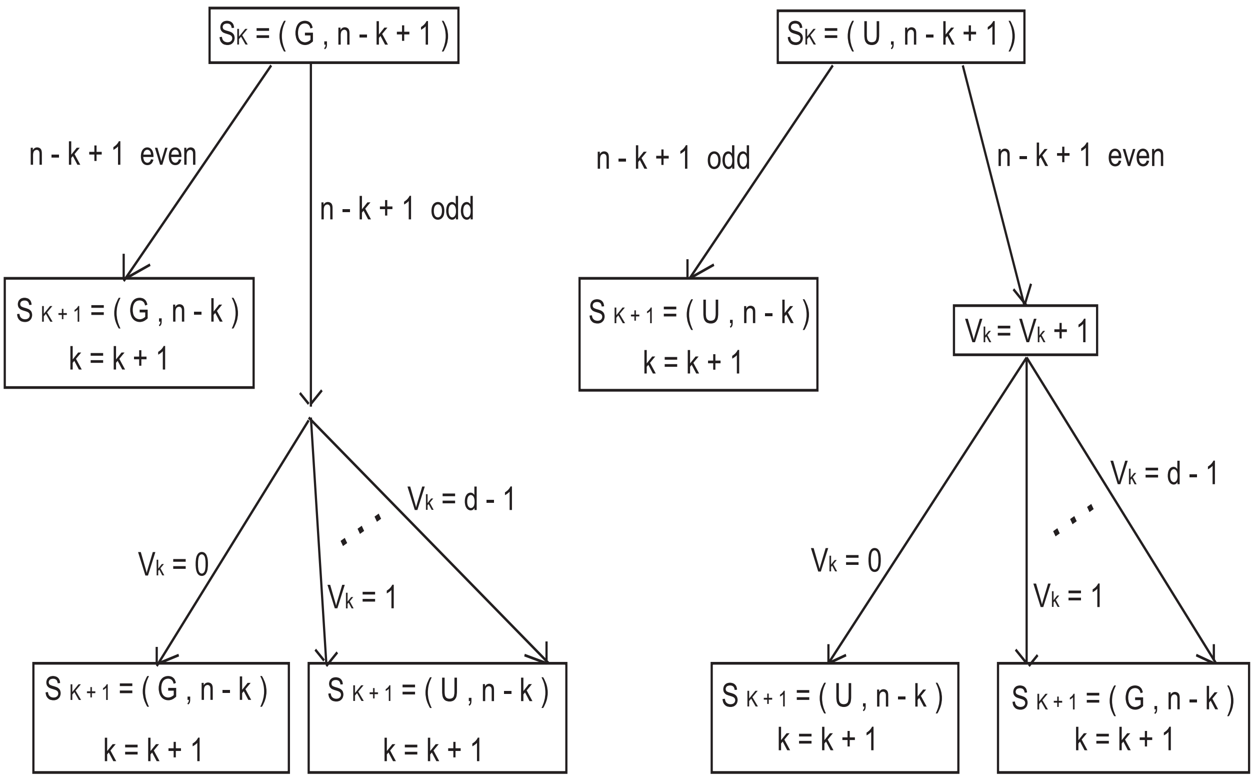

FIGURE 17. A visual representation of Part 1 of the forward algorithm.

FORWARD ALGORITHM: PART 2 (Translates the new $(v_1, v_2, \ldots, v_{n-1})$, which is in Relative coordinates, into RSU coordinates)

$R_1 = (r_1^1 , \ldots, r_n^1) = (v_1 , \ldots, v_{n-1} , 0)$ [2]

$k = 1$

WHILE $k < n$

IF $s_k = (G,\ n-k+1)$ where $n-k+1$ is even

$$R_{k+1} = R_k + (\underbrace{0 , \ldots, 0}_{k\ times} , \underbrace{r_k^1 , r_k^1 , \ldots, r_k^1}_{(n-k)\ times})$$

IF $s_k = (U,\ n-k+1)$ where $n-k+1$ is odd

[2] Note: the superscript on r_i is an index, not a power.

$$R_{k+1} = R_k + (\underbrace{0,\ldots,0}_{k\ times},\underbrace{r_k^1,r_k^1,\ldots,r_k^1}_{(n-k)\ times})$$

ELSE

$R_{k+1} = R_k$

$k = k+1$

END WHILE

RETURN R_n

6.3. **A Proof of the Forward Algorithm.**

We make a few concepts precise before beginning the proof of the algorithm. The proof will consist of three lemmas which together imply that the algorithm returns a codevertex in RSU coordinates.

Definition 6.3.1. *The **top** K_d^{m-1} **subgraph** in a copy of G_d^m or U_d^m is the copy of K_d^{m-1} containing the top vertex of G_d^m or U_d^m, where "top vertex" is taken in the sense of the G-U construction of the SF labeling.*

We will need to consider four types of subgraphs in K_d^n:

- G_d^m, where m is odd.

  The top K_d^{m-1} subgraph (labeled 0) is G_d^{m-1}. Subgraphs 1 through $d-1$ are copies of U_d^{m-1} which are all oriented the same way as G_d^m.

- G_d^m, where m is even.

  Subgraphs 0 through $d-1$ are copies of G_d^{m-1}. Copy i is rotated $\frac{2\pi i}{d}$ radians counterclockwise with respect to the orientation of G_d^m.

- U_d^m, where m is odd.

  Subgraphs 0 through $d-1$ are copies of U_d^{m-1}. Copy i is rotated $\frac{2\pi i}{d}$ radians counterclockwise with respect to the orientation of U_d^m.

- U_d^m, where m is even.

  The top K_d^{m-1} subgraph (subgraph 0) is U_d^{m-1}. Subgraphs 1 through $d-1$ are copies of G_d^{m-1} which are all oriented the same way as U_d^m.

Definition 6.3.2. *The **Relative coordinates** of a vertex in K_d^n are obtained as follows. Suppose the vertex is contained in a certain G_d^m or U_d^m subgraph of K_d^n. Label the K_d^{m-1} subgraphs of this G_d^m or U_d^m with the numbers 0 through $d-1$, where 0 is the top copy (see Definition 6.3.1) and we count counterclockwise. If our vertex is contained in copy i, then the $(n-m+1)^{\text{st}}$ Relative coordinate is i.*

Remark 6.3.3. *The difference between RSU coordinates and Relative coordinates is that Relative coordinates implicitely contain the subgraph rotations inherent in the G-U construction, while RSU coordinates are not aware of the G-U construction.*

Lemma 6.3.4. *If we apply Part 1 of the forward algorithm to every element in the image set* $\{\Phi(m) \mid 0 \le m \le c_n - 1\}$*, then we obtain the correct number of codevertices in each type of subgraph. For example, if we let* $d = 5$ *and* $n = 3$*, then we will obtain* $(\ (G,3)\ ,\ (G,2)\ ,\ (G,1)\ )$ *five times and* $(\ (G,3)\ ,\ (U,2)\ ,\ (G,1)\ )$ *sixteen times.*

Proof

Part 1 is simply a concrete representation of the four cases listed after Definition 6.3.1, except for a small adjustment when $s_k = (U, n-k+1)$ where $n-k+1$ is even.

We have to make this adjustment because the number of codevertices in U_d^{n-k+1} $(n-k+1$ even) is congruent to -1 (mod d), so for $i \in \{0, \ldots, d-2\}$, Φ returns $v_k = i$ once more than it returns $v_k = d-1$. But subgraphs 1 through $d-1$ of U_d^{n-k+1} (all copies of G_d^{n-k}) each contain one more codevertex than subgraph 0 (a copy of U_d^{n-k}).

The adjustment (adding 1 to v_k) compensates for this before the algorithm is allowed to move on, so that we end up with the correct number of vertices in all the copies of G_d^{n-k} and U_d^{n-k}.

There is no such problem with G_d^{n-k+1} $(n-k+1$ odd) since in this case, the *top* K_d^{n-k} subgraph is the one with one more codevertex than the others.

There is no such problem with G_d^{n-k+1} $(n-k+1$ even) and U_d^{n-k+1} $(n-k+1$ odd) because all K_d^{n-k} subgraphs have the same number of codevertices. ■

Lemma 6.3.5. *The vector* R_1 *at the beginning of Part 2 gives the Relative coordinates of a codevertex.*

Proof By Lemma 6.3.4 and the definition of Relative coordinates, the vector $(v_1, v_2, \ldots, v_{n-1})$ returned by Part 1 gives the first $n-1$ components of the Relative coordinates of a codevertex. Since each codevertex is the top vertex of its respective copy of G_d^1, we let the n^{th} coordinate equal zero, so that $R_n = (v_1, \ldots, v_{n-1}, 0)$ gives the Relative coordinates of a codevertex. ■

Lemma 6.3.6. *Part 2 of the forward algorithm takes in the Relative coordinates of a vertex and rotates each* K_d^m *subgraph the correct number of times so as to return the RSU coordinates of the same vertex.*

Proof (See Remark 6.3.3 for an explanation of why rotations are the issue here.)

Suppose we have a K_d^m graph in RSU coordinates. Then when the graph is rotated counterclockwise by $\frac{2\pi}{d}$ radians, a vertex whose coordinates were $(u_1, \ldots, u_m)$ now has coordinates $(u_1+1, \ldots, u_m+1)$.

Now, a K_d^m subgraph in K_d^n has been rotated counterclockwise by $\frac{2\pi}{d}$ radians a total of

$$\gamma = \sum_{i \in B} r_i^1$$

times, where B is the set

$$\{i \le n-m \mid s_i \in \{(G\ ,\ even)\ ,\ (U\ ,\ odd)\}\ \}.$$

Relative coordinates do not show this rotation. So in order to "undo" the rotation of this subgraph, we need to adjust the Relative coordinates of each vertex in our K_d^m subgraph (i.e., adjust w_{m+1} through w_n) by adding γ to each $w_{m+1} \ldots w_n$. Part 2 performs this adjustment one rotation at a time. ■

Theorem 6.3.7. *The forward algorithm takes in $\Phi(m)$ and returns the corresponding codevertex in RSU coordinates.*

Proof The theorem is a direct consequence of Lemmas 6.3.4, 6.3.5, and 6.3.6. ■

6.4. **Inverse Algorithm.**

In this section we provide an inverse to the algorithm described in Section 6.2. The inverse algorithm takes in the RSU coordinates of a codevertex and returns $\Phi(m)$. Like the forward algorithm, we break the inverse algorithm into two parts.

Part 1 of the inverse algorithm inverts Part 2 of the forward algorithm (this is Lemma 6.5.1).

INVERSE ALGORITHM: PART 1 (Changes from RSU coordinates to Relative coordinates. Figure 18 is a visual representation of this part of the algorithm.)

$Y_1 = (y_1^1, \ldots, y_n^1) =$ the given RSU coordinates. [3]
$k = 1$
$g_1 = (G, n)$

WHILE $k < n$

IF $g_k = (G,\ n-k+1)$

IF $n-k+1$ is even

$$g_{k+1} = (G,\ n-k)$$
$$Y_{k+1} = Y_k - (\underbrace{0, \ldots, 0}_{k\ times}, \underbrace{y_k^k, \ldots, y_k^k}_{(n-k)\ times})$$

ELSE

IF $y_k^k = 0$

$$g_{k+1} = (G,\ n-k)$$
$$Y_{k+1} = Y_k$$

ELSE

$$g_{k+1} = (U,\ n-k)$$
$$Y_{k+1} = Y_k$$

ELSE ($g_k = (U,\ n-k+1)$)

IF $n-k+1$ is even

IF $y_k^k = 0$

$$g_{k+1} = (U,\ n-k)$$
$$Y_{k+1} = Y_k$$

ELSE

[3] Note: the superscript on y_i is an index, not a power.

$g_{k+1} = (G,\ n-k)$
$Y_{k+1} = Y_k$

ELSE

$g_{k+1} = (U,\ n-k)$
$Y_{k+1} = Y_k - (\underbrace{0,\ldots,0}_{k\ times}, \underbrace{y^k_k,\ldots,y^k_k}_{(n-k)\ times})$

$k = k+1$

END WHILE

RETURN $Y_n = (y^n_1,\ldots,y^n_n)$

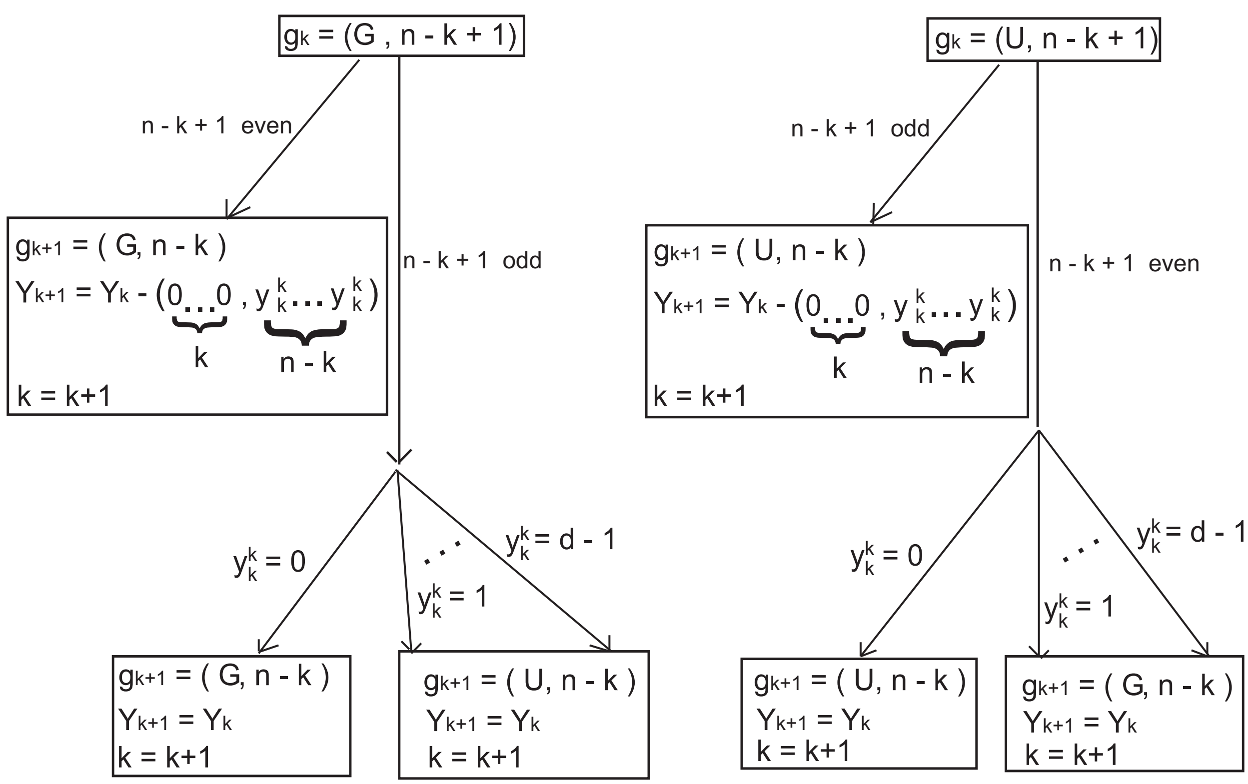

FIGURE 18. A visual representation of Part 1 of the inverse algorithm.

Part 2 of the inverse algorithm takes in Y_n, which is in relative coordinates, and returns the corresponding vector of the form $\Phi(m)$. Therefore this part inverts Part 1 of the forward algorithm (this is Lemma 6.5.2). In fact, it is essentially the same as Part 1 of the forward algorithm, with the exception that when $s_k = (U, n-k+1)$, with $n-k+1$ even, we subtract 1 from v_k instead of adding 1; also, this algorithm returns $(v_1, v_2, \ldots, v_{n-1})$ instead of S.

INVERSE ALGORITHM: PART 2 (Changes from Relative coordinates to a vector of the form $\Phi(m)$)

$(v_1, v_2, \ldots, v_{n-1}) = (y_1^n, \ldots, y_{n-1}^n)$
$k = 1$
$s_1 = (G, n)$

WHILE $k < n$

IF $s_k = (G,\ n-k+1)$

IF $n-k+1$ is even

$s_{k+1} = (G,\ n-k)$

ELSE

IF $v_k = 0$

$s_{k+1} = (G,\ n-k)$

ELSE

$s_{k+1} = (U,\ n-k)$

ELSE (i.e. $s_k = (U,\ n-k+1)$)

IF $n-k+1$ is even

$v_k = v_k - 1$

IF $v_k = 0$

$s_{k+1} = (U,\ n-k)$

ELSE

$s_{k+1} = (G,\ n-k)$

ELSE

$s_{k+1} = (U,\ n-k)$

$k = k+1$

END WHILE

RETURN $(v_1, v_2, \ldots, v_{n-1})$

$\Phi(m) = (v_1, v_2, \ldots, v_{n-1})$

6.5. **A Proof of the Inverse Algorithm.**

Lemma 6.5.1. *Part 1 of the inverse algorithm inverts Part 2 of the forward algorigthm.*

Proof Let $1 \leq p_1 \leq \ldots \leq p_u \leq n$ be all the numbers such that $s_{p_i} \in \{(G,\ even), (U,\ odd)\}$. Then, according to Part 2 of the forward algorithm,

$$R_n = R_1 + \sum_{j=1}^{u} (\underbrace{0\ ,\ 0\ , \ldots,\ 0}_{p_j}\ ,\ \underbrace{r_{p_j}^{p_j}\ ,\ r_{p_j}^{p_j}\ , \ldots,\ r_{p_j}^{p_j}}_{n-p_j})$$

We want to show that $Y_n = R_1$, i.e., that

$$Y_n = R_n - \sum_{j=1}^{u} (\underbrace{0\ ,\ 0\ , \ldots,\ 0}_{p_j}\ ,\ \underbrace{r_{p_j}^{p_j}\ ,\ r_{p_j}^{p_j}\ , \ldots,\ r_{p_j}^{p_j}}_{n-p_j}).$$

Note that $R_1 = \ldots = R_{p_1}$ and $Y_1 = \ldots = Y_{p_1}$. Furthermore, $r_i^{p_1} = r_i^1 = y_i^{p_1}$, since we have that $Y_1 = R_n$ and $r_i^{p_1} = r_i^n$ for $i = 1 \ldots p_1$. So $s_i = g_i$ for all $i = 1 \ldots p_1 + 1$.

Thus, the first vector that gets subtracted from Y_1 in Part 2 of the inverse algorithm is

$$(\underbrace{0\ ,\ 0\ , \ldots,\ 0}_{p_1}, \underbrace{y_{p_1}^1\ ,\ y_{p_1}^1\ , \ldots,\ y_{p_1}^1}_{n-p_1}) = (\underbrace{0\ ,\ 0\ , \ldots,\ 0}_{p_1}, \underbrace{r_{p_1}^1\ ,\ r_{p_1}^1\ , \ldots,\ r_{p_1}^1}_{n-p_1})$$

Now Y_{p_1+1} and R_1 have the first p_2 entries identical, and S and G have the first $p_2 + 1$ entries identical, so the next vector which gets subtracted from Y_{p_1+1} is

$$(\underbrace{0\ ,\ 0\ , \ldots,\ 0}_{p_2}, \underbrace{r_{p_2}^1\ ,\ r_{p_2}^1\ , \ldots,\ r_{p_2}^1}_{n-p_2})$$

and so on. ■

Lemma 6.5.2. *Part 2 of the inverse algorithm inverts Part 1 of the forward algorithm.*

Proof This is clear, since the two algorithms are identical except that one contains "$v_k = v_k + 1$" while the other contains "$v_k = v_k - 1$" ■

Theorem 6.5.3. *The inverse algorithm inverts the forward algorithm.*

Proof Direct consequence of Lemmas 6.5.1 and 6.5.2. ■

7. Encoding and Decoding for the SF Labeling

In this section, we present an encoding and decoding scheme for the SF labeling.

Definition 7.0.4. *An* ***encoding and decoding scheme*** *for a particular labeling of K_d^n is a bijection between the set of natural numbers $\{\ 0\ ,\dots,\ c_n - 1\ \}$, where c_n is the number of codevertices in a P1ECC on K_d^n, and the set of codewords in the labeling.*

Note that, unlike the results from Sections 5 and 6, an encoding and decoding scheme is designed for one particular labeling method.

7.1. **Toward an Encoding Scheme.**

Cull and Nelson present an easy encoding and decoding scheme for the case when $d = 3$. Their scheme uses the fact that in the $d = 3$ case, every distance 1 neighborhood of a codeword contains exactly one word which is a multiple of 4 (when viewed as a number in base 3). They denote by G_n the set of codewords in the SF labeling of K_3^n. To encode the number $m < |G_n|$, one simply runs the number $4m$ through the finite state machine for error correction. [3]

We asked if Cull and Nelson's technique could be easily generalized for $d > 3$ (using multiples of $d+1$ instead of multiples of 4). Unfortunately, the answer is no. This is because in the general case, some codewords are not adjacent to any multiple of $d+1$, while others are adjacent to two multiples of $d+1$. An easy example is K_5^3, where the codeword 414 is adjacent to 314 and 424, both of which are multiples of 6.

After trying some similar ideas and failing, we decided to design an encoding and decoding scheme which would use few (if any) theoretical properties of the SF labeling. In particular, we wrote two algorithms, discussed in detail in Sections 5 and 6, which together give a bijection between $\{\ 0\ ,\dots,\ c_n - 1\ \}$ and the set of RSU coordinates of codevertices. Now all that is left is to find a simple bijection between RSU coordinates of codevertices and the codewords of the SF labeling. Fortunately, this turns out to be fairly straightforward.

Note: The advantage of this approach is that our bijection between natural numbers and RSU coordinates has nothing to do with any labeling of K_d^n. Furthermore, any reasonable labeling has an easy map between the position of a vertex and its label. Therefore, our technique could be imitated to create encoding and decoding schemes for most labelings.

7.2. **The SF Labeling as a Tree.**

In this section we give an intuitive motivation for our encoding and decoding scheme. We show how one can view the construction of the SF labeling as a d-ary tree, where each “layer” of the tree represents the SF labeling of K_d^m for a particular m (this makes sense because the SF labeling is constructed recursively). Each node in the m^{th} layer represents a vertex in K_d^m and is labeled with the SF label of that vertex as well as its RSU coordinates. The encoding and decoding scheme passes between these two pieces of information by traveling up the tree through one type of information, turning around at the root, and travelling back down to the same vertex through the other type of information.

Suppose we have the SF labeling of K_d^m. Then the SF labeling of K_d^{m+1} is constructed by applying α to the labels in K_d^m, creating d copies of the graph, rotating each an appropriate number of times, and appending the appropriate digit to all the labels in each (see Section 3.1). So if $b_1 \ldots b_m$ is the label of a vertex in K_d^m, then $b_1 \ldots b_m$ gives rise to d “daughter” labels in K_d^{m+1}. These labels are $\alpha(b_1 \ldots b_m) \circ i$ where $i \in \{0, \ldots, d-1\}$. So we can think of the construction of the SF labeling as a d-ary tree where each vertex in K_d^m is the root of a subtree of the form in Figure 19.

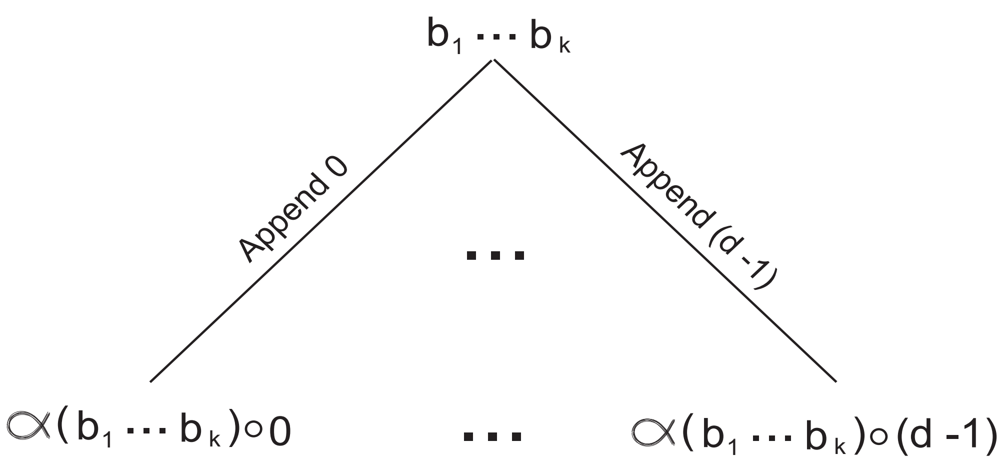

FIGURE 19. A vertex label in K_d^m and its d “daughter” labels.

As we said above, the SF labeling makes d copies of $\alpha(K_d^m)$ and rotates the i^{th} copy $\frac{2\pi i}{d}$ radians clockwise before connecting all the copies by edges. So the i^{th} daugter of the vertex whose RSU coordinates in K_d^m were $(\, a_1 \,, \ldots, \, a_m \,)$, is the vertex in K_d^{m+1} with RSU coordinates $(\, i \,, \, a_1 - i \,, \ldots, \, a_m \,)$. Figure 20 is the same as Figure 19, with the addition that each node is also labeled with the RSU coordinates of the corresponding vertex.

Our encoding and decoding scheme is the composition of the algorithm from Sections 5 and 6, and the algorithm we give in Section 7.3, which is a bijection between SF labels of codevertices and RSU coordinates of codevertices. Note that these last two types of data are shown in Figure 20. The technique will be to specify one type of data for a node in the tree, then follow a path up to the root by computing this same data for all the nodes along the path. One then specifies the other type of data for the last node in the path, then follows the reverse path back down by computing the new data for each node, finally arriving at the new data for the original node.

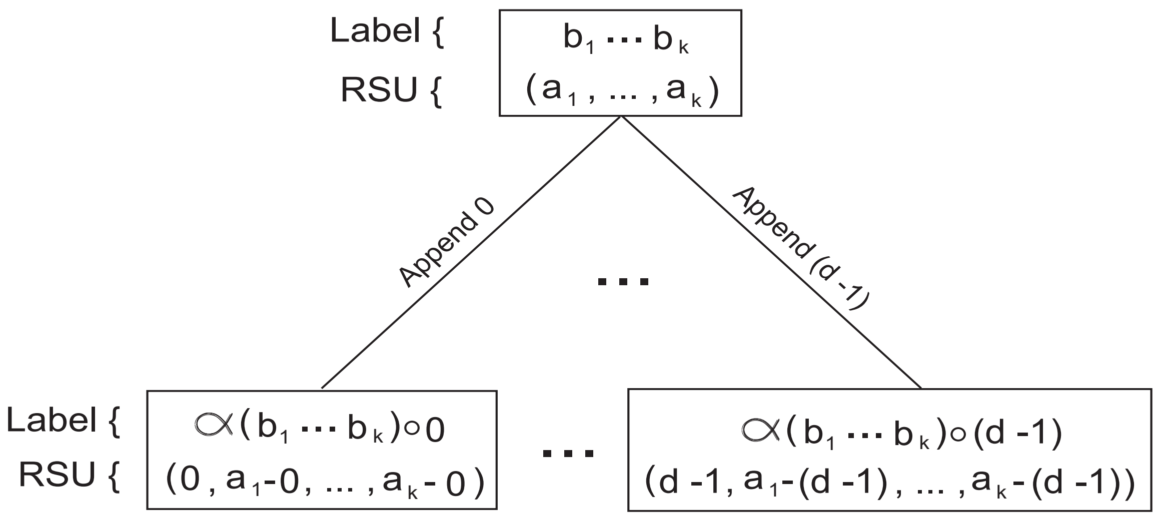

FIGURE 20. Figure 19 with RSU coordinates added.

7.3. **Encoding and Decoding Algorithms.**

Recall that the forward algorithm (Section 6) actually gives a bijection between the set of vectors of the form $\Phi(m)$ and the set of RSU coordinates of codevertices. Call this map F. So the map $\Lambda = F \circ \Phi$ is a bijection between the first c_n natural numbers and the RSU coordinates of codevertices. Our encoding scheme is the map *ENCODE* from RSU coordinates of codevertices to codewords. The decoding scheme, *DECODE*, is the inverse of *ENCODE*.

ALGORITHM: *ENCODE*: PART 1 [4]

$$Q_1 = (\, q_1^1 \,, \ldots, \, q_n^1 \,) = \Lambda(m)$$

$$k = 1$$

WHILE $k < n$

$$\begin{aligned} Q_{k+1} &= (\, q_1^{k+1} \,, \ldots, \, q_n^{k+1} \,) \\ &= (\, q_2^k \,, \ldots, \, q_{n-k+1}^k \,) + (\, \underbrace{q_1^k \,, \ldots, \, q_n^k}_{n-k} \,) \end{aligned}$$

$$k = k + 1$$

END WHILE

ENCODE: PART 2

$$y_1 = q_1^n$$

$$k = 1$$

[4]Note: In both *ENCODE* and *DECODE*, the superscript on q_i is an index, not a power. Also, in *ENCODE*: PART 1, the Q_i's are vectors of different lengths; in particular, Q_i has one more component than Q_{i+1}.

WHILE $k < n$

$y_{k+1} = \alpha(y_k) \circ q_1^{n-k}$

$k = k + 1$

END WHILE

RETURN y_n

$ENCODE(\Lambda(m)) = y_n$

(where $\alpha(y_k)$ means apply the permutation α to the digits of the string y_k in order, producing a new string, and $string1 \circ string2$ means append $string2$ to $string1$.

ALGORITHM: $DECODE$: PART 1

Given codeword $c_1 \cdots c_n$

$q_1^1 = c_n$

$k = 1$

WHILE $k < n$

$q_1^{k+1} = \alpha^{-k}(c_{n-k})$

$k = k + 1$

END WHILE

$DECODE$: PART 2

FOR $i = 2 \ldots n$

$q_i^1 = q_1^i - \sum_{j=1}^{i-1} q_1^j$

END FOR

RETURN Q_1

7.4. **Proofs of** *ENCODE* **and** *DECODE***.**

To aid in the proof of our encoding and decoding algorithm, Figure 21 gives a more global view of the SF labeling tree. This diagram uses the same notation as *ENCODE* and *DECODE*.

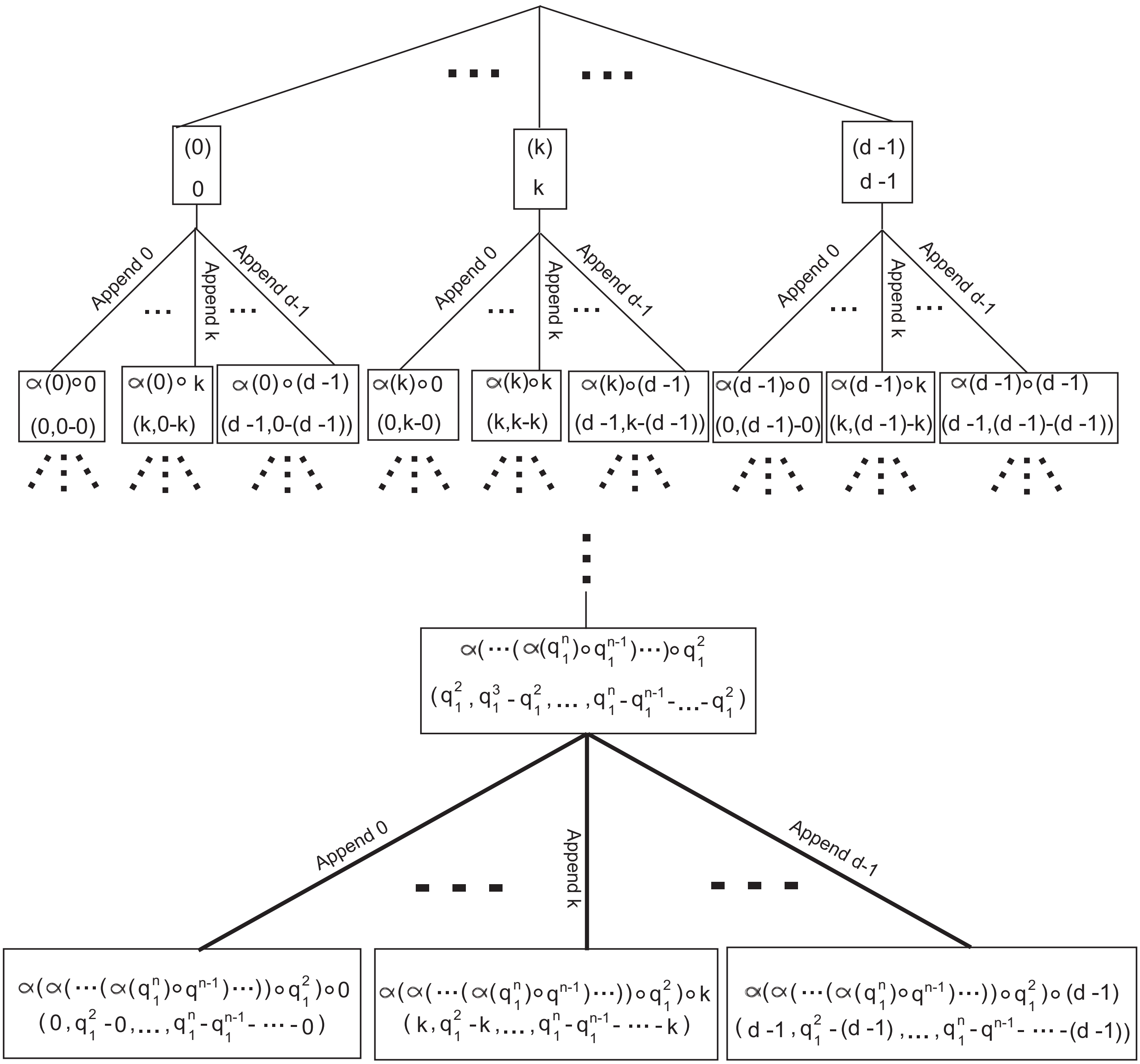

FIGURE 21. Tree representation of the SF labeling, using notation from *ENCODE* and *DECODE*. At each node, the top label gives the SF label and the bottom label gives the RSU coordinates.

Proposition 7.4.1. *ENCODE takes in the RSU coordinates of a vertex in K_d^n and returns the SF label of the same vertex.*

Proof

Suppose we have the RSU coordinates of a vertex v in K_d^n. Part 1 of *ENCODE* computes the RSU coordinates of its "parent" vertex in K_d^{n-1}, then computes the RSU coorinates of the parent vertex of *that* vertex, and so on, up to its ancestor in K_d^1. By inspection of the general subtree in Figure 21, *ENCODE* performs the necessary operation correctly at each step.

Part 2 of *ENCODE* follows the reverse path back down the tree to the original node. Note that any path through the tree is completely determined by which digits are appended to successive labels. The algorithm follows the path which is determined by information gathered in Part 1. It computes α and then appends the appropriate digit. By inspection of the general subtree in Figure 21, this is performed correctly at each step. Part 2, therefore, finishes by returning the label of the original vertex. ■

Proposition 7.4.2. *DECODE takes in the SF label of a vertex in K_d^n and returns the RSU coordinates of the same vertex.*

Proof

The proposition can be proved in a similar fashion to Proposition 7.4.1. ■

8. CONCLUSION

We have presented an encoding and decoding method for the SF labeling of odd-dimension iterated complete graphs. The method uses general properties of P1ECC's on iterated complete graphs; only in the last step does it use any information which is particular to the SF labeling.

Further work could include testing this technique on other labeling schemes to determine if it will be useful as predicted. Work could also be done to simplify the algorithms themselves as well as the proofs.

REFERENCES

[1] Shawn Alspaugh, Nathan Knight, and Kathleen Meloney. Perfect One Error Correcting Codes on Iterated Complete Graphs. Proceedings of the REU Program in Mathematics. NSF and Oregon State University. Corvallis, Oregon. 2001.

[2] Be Birchall and Jason Tedor. Perfect One Error Correcting Codes on Iterated Complete Graphs. Proceedings of the REU Program in Mathematics. NSF and Oregon State University. Corvallis, Oregon. 1999.

[3] Paul Cull and Ingrid Nelson. Perfect Codes, *NP*-Completeness, and Towers of Hanoi Graphs. *Bulletin of the ICA* 26:13-38, 1999.

[4] Christopher Frayer and Shalini Reddy. Perfect One Error Correcting Codes and Iterated Complete Graphs. Proceedings of the REU Program in Mathematics. NSF and Oregon State University. Corvallis, Oregon. 2002.

[5] Stephanie Kleven. Perfect Codes on Odd Dimension Serpinski Graphs. Proceedings of the REU Program in Mathematics. NSF and Oregon State University. Corvallis, Oregon. 2003.

UNIVERSITY OF PENNSYLVANIA
E-mail address: `prussell@math.upenn.edu`