PUZZLES ON GRAPHS: THE TOWERS OF HANOI, THE SPIN-OUT PUZZLE, AND THE COMBINATION PUZZLE

LINDSAY BAUN AND SONIA CHAUHAN

ADVISOR: PAUL CULL
OREGON STATE UNIVERSITY

ABSTRACT. The Towers of Hanoi is a well known puzzle that has been studied for years. The Spin-Out puzzle is a lesser known and less studied puzzle. However, both of these puzzles have a nice correlation to iterated complete graphs. Previous research shows that both the Towers of Hanoi puzzle and the Spin-out puzzle can be represented by a labeling on iterated complete graphs. These puzzles have constructions for their labelings, a Perfect One-Error Correcting Code on their graphs, and algorithms to solve the puzzle. Previous work by Skubak and Stevenson [1] created the combination puzzle, which combines the Towers of Hanoi and the Spin-Out puzzle to create a new puzzle, which corresponds to graphs on even dimensions. We show existing patterns for a recursive labeling construction on graphs corresponding to the Combination puzzle. We also present the iterative, recursive and count algorithms for the SF puzzle which is the extension of the Towers of Hanoi. And we provide the iterative and recursive algorithm for the dimension 2^m, puzzle which is the extension of the Spin-out puzzle. We also observe Encoding an Decoding procedures using Hamiltonian paths for the graphs corresponding to the Towers of Hanoi and the Spin-out puzzle.

Date: 14 August 2009.

Key words and phrases. graphs, codes, puzzles, Towers of Hanoi, Spin-Out, iterated complete graphs, error-correction, encoding and decoding.

This work was done during the Summer 2009 REU program in Mathematics at Oregon State University.

CONTENTS

1. Introduction

The Towers of Hanoi and Spin-out puzzle are interesting puzzles that have existing algorithms. These algorithms include recursive, iterative and count algorithms. These puzzles also generalize to puzzles known as the SF Puzzle and the Dimension 2^m Puzzle, respectively. We show in section 3.3 the methods used to construct the algorithms for the SF puzzle and in section 4.4 the algorithms for the Dimension 2^m Puzzle. The graphs corresponding to the Towers of Hanoi and Spin-out puzzles also exhibit nice properties. These properties include: easy construction of labeling, Gray-code property, codeword recognition, error-correcting machines, encoding and decoding schemes and a finite-state machine to calculate distance between any two configurations. These properties will be useful in coding theory and computer applications. Our research aimed to find some properties in the SF puzzle and/or the Dimension 2^m puzzle that would make their labelings more attractive. We show our results in trying to find a simple encoding and decoding scheme for the SF puzzle and the Dimension 2^m puzzle in section 6. Although the method of using Hamiltonian paths extended to the Towers of Hanoi and the Spin-out puzzles it did not extend to the SF puzzle or the Dimension 2^m puzzle. We give examples of this method and why it would not extend as desired. Lastly, we observed the distance problem which was simple for the Towers of Hanoi but further investigation still needs to be done for the other puzzles.

2. Graphs, Labels, and Codes

This section contains definitions and background information on iterated complete graphs as well as basic definitions of labels and codes on graphs.

2.1. **Iterated Complete Graphs.**

Definition 2.1. *A **(simple) graph** $G = (V,E)$ consists of a finite set $V(G)$ (called **vertices**) and a set $E(G)$ (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 2.2. *The **degree** of a vertex v is the number of vertices which are adjacent to v.*

Definition 2.3. *The **complete graph** on d vertices, denoted K_d, is the graph such that all the vertices are pairwise adjacent. That is, $|V(K_d)| = d$ and $E(K_d) = \{\text{unordered pairs } (a,b) : a,b \in V(K_d)\}$.*

Figure 1 shows three complete graphs.

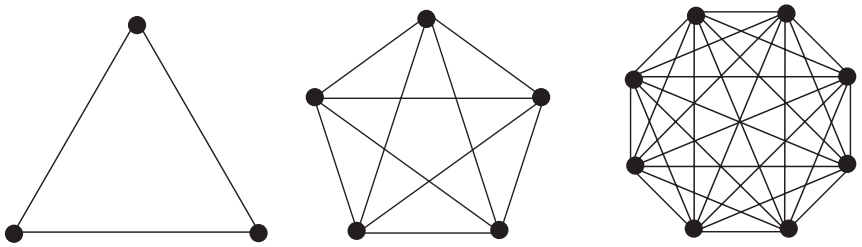

FIGURE 1. The complete graphs K_3, K_5, and K_8.

Definition 2.4. *An **iterated complete graph** on d vertices with n iterations, denoted K_d^n, can be defined recursively. K_d^1 is the complete graph on d vertices. K_d^n is composed of d copies of K_d^{n-1} and edges such that exactly one edge connects each K_d^{n-1} subgraph to every other K_d^{n-1} subgraph and exactly one vertex in each of the K_d^{n-1} subgraphs has degree $d-1$.*

We say that a graph K_d^n has **dimension** d.

Definition 2.5. *A **subgraph** M of a graph G consists of a subset $V(M) \subset V(G)$ together with the associated edges.*

In particular, the d copies of K_d^{n-1} from which K_d^n is constructed are all subgraphs of K_n^d.

The graph K_d^n can simply be thought of as d copies of K_d^{n-1} connected in a nice way, or alternatively as the graph K_d^{n-1} with each vertex replaced by a copy of K_d. Figure 2 shows the graphs K_5^1, K_5^2 and K_5^3, illustrating how each graph is constructed from the graph of the previous dimension.

Definition 2.6. *A **corner vertex**, or simply **corner**, of the graph K_d^n is a vertex with degree $d-1$. A **non-corner vertex** is simply a vertex that is not a corner. All non-corner vertices of iterated complete graphs have degree d.*

2.2. **Codes on Graphs.**

Definition 2.7. *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 $v \notin C$.*

Definition 2.8. *A code in a graph is **error-correcting** if for every vertex v there is a unique $C(v) \in C$ so that $C(v)$ is the codeword closest to v.*

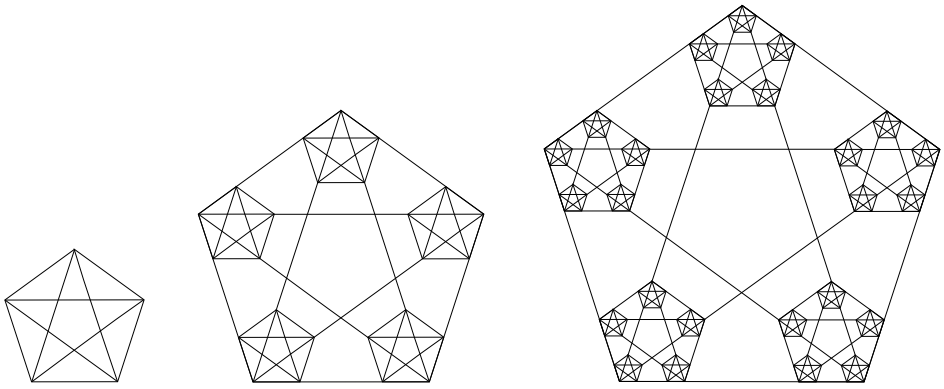

FIGURE 2. The iterated complete graphs K_5^1, K_5^2 and K_5^3.

Definition 2.9. *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.*

2.3. **Labelings and Codewords.**

Definition 2.10. *A **labeling** on K_d^n is a method of assigning strings to the vertices of K_d^n such that this method gives a bijection between vertices and strings. The string assigned to a vertex will be called the **label** of that vertex.*

Definition 2.11. *In a labeling of G, a **codeword** is the label of a codevertex. A **noncodeword** is the label of a noncodevertex.*

We say that L_d^n is the labeling of K_d^n. Which labeling we mean will be clear from the context.

Definition 2.12. ***Coding** and **Decoding** are the mappings between the integers and the strings of codewords (codestrings), and vice-versa. That is there exists two mappings so that:*

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

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

Where G_n is the set of codestrings.

Definition 2.13. *Let G be a graph. A labeling of G has the **Gray code property** if every pair of adjacent vertices has labels which differ in exactly one position.*

2.4. **The G-U Construction.** Cull and Nelson [7] proved that determining whether a given graph has a P1ECC is an NP-complete (difficult) problem. However, they introduced a relatively simple method for constructing a P1ECC on K_3^n for any iteration n. Also, they proved that this code is unique up to rotation, with strict uniqueness if a specified corner of K_3^n is required to be a codeword. These results were also found to generalize to higher dimensions. Cull and Nelson's method has come to be known as the G-U construction.

The G-U construction uses two types of codes on K_d^n: G-codes and U-codes. 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 3 shows G_5^1 and U_5^1.

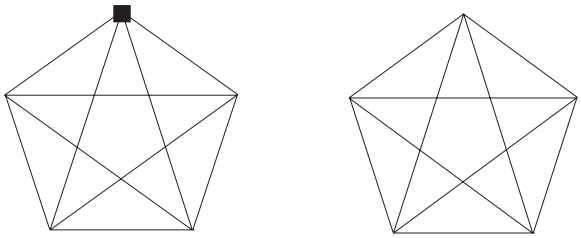

FIGURE 3. G_5^1 and U_5^1.

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 so 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 non-top 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 non-top corner vertex in one copy to a non-top corner vertex in the other copy.
    - Exactly one non-top 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 non-top 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 non-top corner vertex in one copy to a non-top corner vertex in the other copy.
    - Exactly one non-top 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 4 shows G_5^2 and U_5^2. Figure 5 shows G_5^3 and U_5^3.

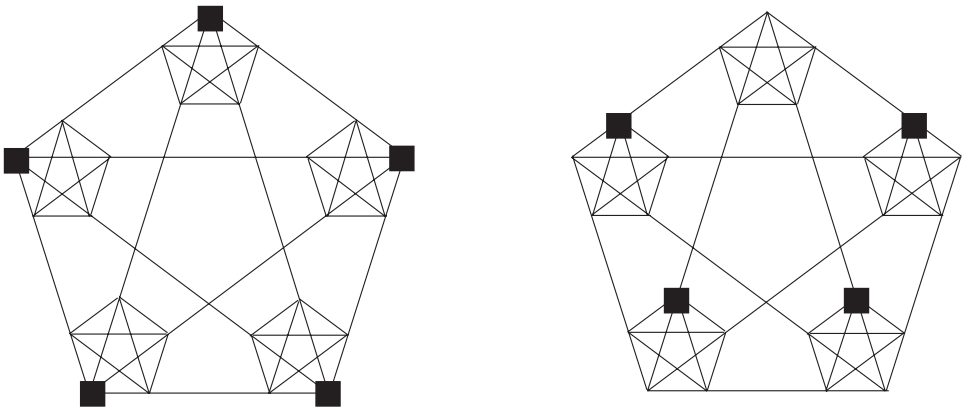

FIGURE 4. G_5^2 and U_5^2.

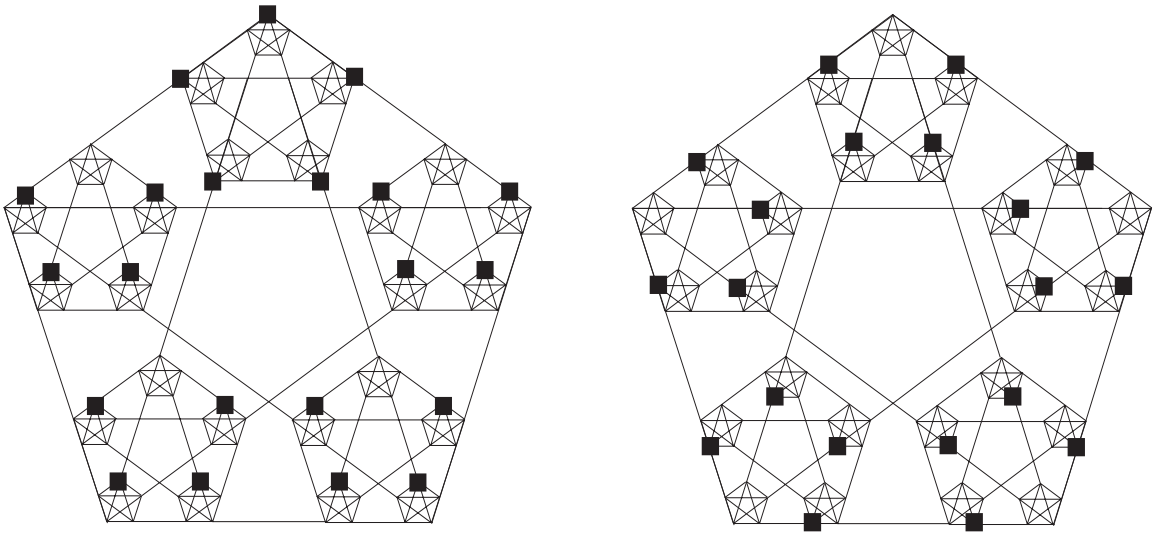

FIGURE 5. G_5^3 and U_5^3.

3. THE SF LABELING AND PUZZLE

In previous papers, labels and puzzles have been established for odd dimensional graphs. [4] The SF labeling on the odd dimensional iterated complete graphs has been established to have finite-state machines for codeword recognition and error correction. The SF labeling also has the Gray code property and corresponds to a puzzle called the SF puzzle. In the case $d = 3$, the SF labeling corresponds to the Towers of Hanoi labeling given by Cull and Nelson [7]. It has been demonstrated that even dimensional iterated complete graphs do not support SF-like labelings.

3.1. **Construction of the SF Labeling.** 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 6 shows the SF labeling of K_5^1.

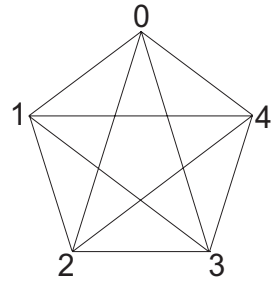

FIGURE 6. The SF labeling of K_5^1.

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 clockwise, then append k to each word in this copy. Finally, connect the d copies to form K_d^n. Figure 7 shows the SF labeling of K_5^3.

3.2. **The SF Puzzle.** The Towers of Hanoi is played with n disks all of different size. The disks are stacked on three towers so that no larger disk is stacked on top of a smaller one. The goal is to begin with all disks on one tower and move them to another. We will number the towers 0, 1, and 2. A natural way to label the configurations of disks on towers is with ternary strings as follows. Record the tower number of the smallest disk. To the right of this number, record the tower number of the next smallest disk. Continue in this way to obtain a string of length n. Now each vertex of K_3^n has an SF label that corresponds to a configuration of the Towers of Hanoi puzzle. The labels of adjacent vertices represent configurations which are one legal move from each other. Figure 8 shows the SF labeled graph K_3^3 corresponding to Towers of Hanoi with 3 disks.

For an odd number $d \geq 3$ of towers numbered 0 through $d-1$ the configurations of n disks on these towers can be represented by base d strings of length n. This puzzle is known as the SF puzzle. The SF puzzle has the same rules as Towers of Hanoi:

(1) Only one disk is moved at a time.

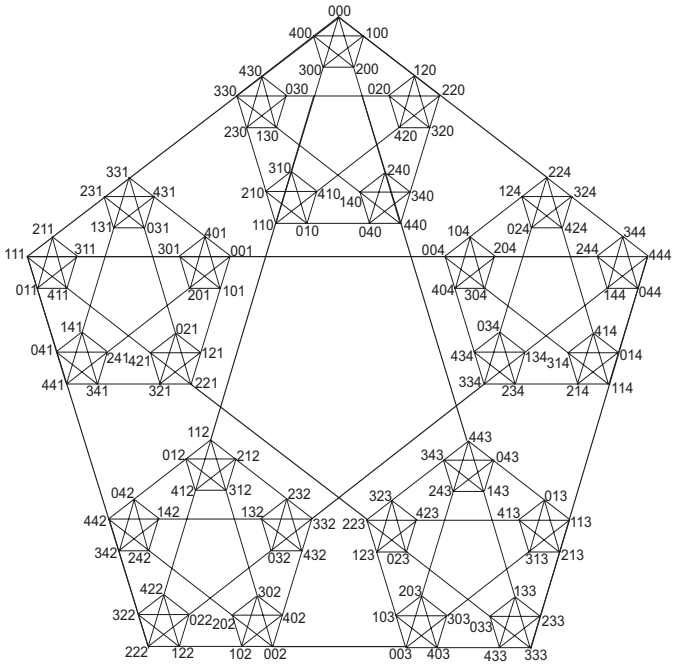

FIGURE 7. The SF labeling of K_5^3.

(2) A larger disk is never placed on top of a smaller disk.

In addition, The SF puzzle has the following restrictions to guarantee the puzzle satisfies the SF labeling:

(1) No disk may be moved unless all of the disks smaller than it are stacked together on the same tower.
(2) When a disk is able to move, if the stack of smaller disks is on tower a and the disk to be moved is on tower b, then the disk may only move to tower $(2a-b) \bmod d$.

Observe the smallest disk is able to move to any tower because it is unaffected by these rules. Figure 9 shows configurations corresponding to labels 220 and 224 on K_5^3. Here the largest disk can only move between towers 0 and 4, and there is an edge between these two vertices in K_5^3, as shown in Figure 7.

3.3. **Algorithms to Solve The SF Puzzle.** For the SF puzzle, with three towers, Cull and Ecklund [10] have shown there exists a recursive, iterative and count algorithm to solve the puzzle. These algorithms also exist for the general case with $d \geq 3$ and odd. Recall the rules of the SF puzzle along with the necessary restrictions presented in section 3.2. And again observe

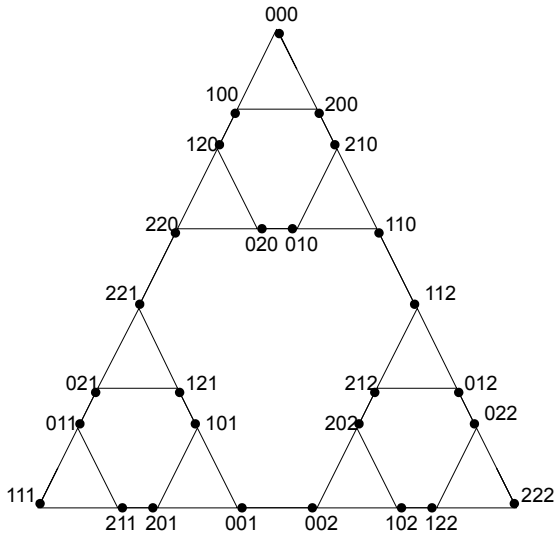

FIGURE 8. The labeled graph K_3^3 corresponding to the Towers of Hanoi with 3 disks.

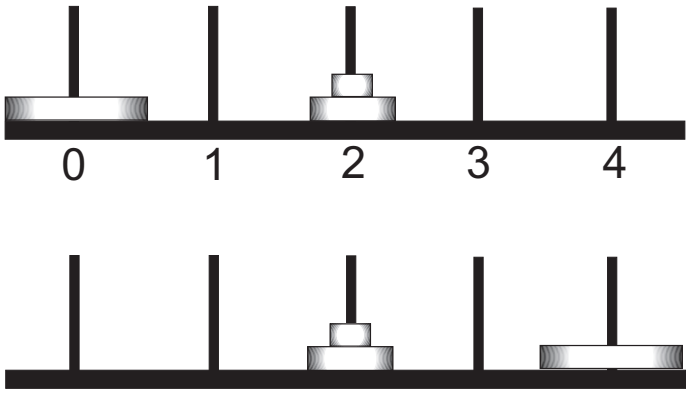

FIGURE 9. Configurations corresponding to labels 220 and 224 on K_5^3. The largest disk may move between towers 0 and 4.

that the smallest disk is able to move to any tower because it is unaffected by these rules.

3.3.1. *Recursive Algorithm.* The goal of the SF puzzle is to move all n disks from tower 0 to tower $d-1$. Define the first tower as 0 and the second tower as 1 and so on until the last tower, $d-1$. The largest disk can only move to tower $d-1$ if all the smaller disks are stacked together on tower

a where $(2a-0) \bmod d = d-1$. In the algorithm, rather than starting at tower 0 and going to tower $d-1$, the algorithm will start with tower i and end at tower j. This will make the algorithm more general. Therefore, the $n-1$ disks must move to tower a where $(2a-i) \bmod d = j$. From these observations we come to the recursive algorithm:

PROCEDURE HANOI(i, j, n)
IF $n = 1$ THEN move the top disk from tower i to tower j
ELSE HANOI $(i, [(i+j)2^{-1}] \bmod d, n-1)$
move the top disk from tower i to tower j
HANOI$([(i+j)2^{-1}] \bmod d, j, n-1)$

Where the inputs i and j represent source and destination, respectively, and n is a number indicating the number of disks that will move from the source to the destination.

Proposition 3.1. *The recursive algorithm HANOI correctly solves the Towers of Hanoi problem.*

Because this proof will be done by induction we will define an inductive hypothesis. Notice this proof is very similar to the proof of the standard Towers of Hanoi recursive algorithm.

Definition 3.2. *Define: HYP(n) = HANOI(i, j, n) correctly moves the n disks $1, 2, ..., n$ from tower i to tower j.*

In addition we also need i and $j \in \{0, 1, 2, ..., d-1\}$. When we state *correctly* we mean that the rules and restrictions, mentioned in section 3.2, are obeyed.

Proof. BASE CASE: Consider HYP(1) which we will show is**TRUE**. From the definition, HYP(1) says that HANOI$(i, j, 1)$ *correctly* moves the first disk from tower i to tower j. In the algorithm the **IF** condition holds because $n = 1$ and therefore executes the **THEN** condition which moves the top disk from tower i to tower j. Observe this moves disk 1 from tower i to tower j. We still need to show this *correctly* moves disk 1. Notice that disk 1 is able to *correctly* move to any tower and therefore satisfies all the rules. After moving disk 1 HANOI runs out of instructions and terminates. Thus, HYP(1) is **TRUE**.

INDUCTIVE STEP: We want to show HYP$(n-1)$ implies HYP(n) for all $n > 1$. Assume HYP$(n-1)$ is **TRUE**. Consider the algorithm HANOI with the inputs (i, j, n) where $n > 1$. Because the **IF** condition is false we move onto the **ELSE** condition of the algorithm. The **ELSE** condition tells us to

call HANOI$(i,[(i+j)2^{-1}] \bmod d, n-1)$. Because we assumed HYP$(n-1)$ is true we know HANOI$(i,[(i+j)2^{-1}] \bmod d, n-1)$ *correctly* moves the $n-1$ disks $1,2,...,n-1$ from tower i to tower $[(i+j)2^{-1}] \bmod d$.

The next instruction is to move the top disk from tower i to tower j. Notice the top disk of tower i is disk n, the largest disk, which we will show *correctly* moves to tower j. We will show that the rules of the game are being obeyed. Rule (1) is obeyed because only disk n is being moved. Because the disks smaller than disk n are placed on tower $[(i+j)2^{-1}] \bmod d$, Rule (2) is obeyed. Now we show the restrictions are obeyed. Restriction (1) states all the smaller disks are stacked together on one tower, which is true since disks $1,2,....,n-1$ are all on tower $[(i+j)2^{-1}] \bmod d$. Restriction (2) is a little trickier to show. Observe the smaller disks are on tower $[(i+j)2^{-1}] \bmod d$ and disk n, the disk we want to move, is on tower i. Disk n may only *correctly* move to tower $[2[(i+j)2^{-1}]-i] \bmod d$ in order for Restriction (2) to hold. Using modular arithmetic we easily get:

$$[2[(i+j)2^{-1}]-i] \bmod d = [(i+j)-i] \bmod d = j \bmod d.$$

This proves that Restriction (2) is obeyed since disk n can only move to tower j. Therefore the top disk, disk n, moves from tower i to tower j *correctly*.

The next part of the algorithm is to call HANOI$([(i+j)2^{-1}] \bmod d, j, n-1)$, and by HYP$(n-1)$, HANOI$([(i+j)2^{-1}] \bmod d, j, n-1)$ *correctly* moves disks $1,2,....,n-1$ from tower $[(i+j)2^{-1}] \bmod d$ to tower j. Notice Rule (1) is obeyed since HYP$(n-1)$ is true. By HYP$(n-1)$ none of the disks from among $1,2,...,n-1$ are ever placed on a larger disk from among $1,2,...,n-1$. Since disk n has been moved to tower j, the disks $1,2,...,n-1$ are all smaller than disk n and therefore will not cause Rule (2) to be disobeyed. The restrictions hold for disks $1,2,....,n-1$ and so we only need to show they hold for disk n. Restriction (1) and Restriction (2) hold because they are independent of where disk n is located. Therefore, the algorithm HANOI, with the inputs (i,j,n) where $n>1$, *correctly* moves all n disks from tower i to tower j. By induction, HYP(n) is true for all $n \geq 1$ and so proposition 3.1 holds. □

3.3.2. *Iterative Algorithm.* When we call HANOI(i,j,n) this moves n disks from tower i to j where i and $j \in \{0,1,2,...,d-1\}$. Next, we want to observe the iterative algorithm for $d>3$. For this we will first prove that the smallest disk, called disk 1, will always move at the same increment.

Lemma 3.3. *The recursive algorithm, HANOI, moves disk* 1 *at the same increment for all* $n \geq 1$ *and the increment is solely a function of* $(j-i) \bmod d$.

Proof. BASE CASE: Consider the case where $n = 1$. For HANOI$(i, j, 1)$ the **IF** condition is **TRUE**, so the algorithm executes the **THEN** condition and moves disk 1 from i to j. This is the only move and therefore disk 1 always moves at the same increment.

INDUCTIVE STEP: Assume disk 1 moves at the same increment in the Algorithm HANOI $(i, j, n-1)$. We will show this implies disk 1 moves at the same increment for HANOI (i, j, n). The algorithm for HANOI with the inputs (i, j, n), for $n > 1$, first calls HANOI $(i, [(i+j)2^{-1}] \bmod d, n-1)$. From our assumption this call will always move disk 1 by the same increment, call it I. The next step is to move the largest disk, disk n, which will not change the increment disk 1 is moving. Then we call HANOI $([(i+j)2^{-1}] \bmod d, j, n-1)$ which, by our assumption, will always move disk 1 with the same increment, call it K.

Next we need to show $I = K$. When we call HANOI $(i, [(i+j)2^{-1}] \bmod d, n-1)$ we are simply moving $n-1$ disks from i to $[(i+j)2^{-1}] \bmod d$. In other words the $n-1$ disks moved at the total increment of

$$[(i+j)2^{-1}] \bmod d - i = [(i+j)2^{-1} - (2)(2^{-1})i] \bmod d$$
$$= [2^{-1}(i+j-2i)] \bmod d = [2^{-1}(j-i)] \bmod d$$

Similarly, the next call HANOI $([(i+j)2^{-1}] \bmod d, j, n-1)$ moves the $n-1$ disks at the total increment of

$$[j - (i+j)2^{-1}] \bmod d = [(2)(2^{-1})j - (i+j)2^{-1}] \bmod d$$
$$= [2^{-1}(2j - i + j)] \bmod d = [2^{-1}(j-i)] \bmod d$$

. Therefore, both calls move the $n-1$ disks at the same total increment. But, this is just a relabelling of the moves in which the tower names are cyclically shifted. Hence, $I = K$.

Since both calls always move disk 1 by the same increment we can say disk 1 moves at the same increment for HANOI (i, j, n) and by induction, we conclude disk 1 always move at the same increment for all $n \geq 1$. □

Lemma 3.3 will be helpful in creating an iterative algorithm for the general Towers of Hanoi. For this we need to know the increment in which disk 1 is moving, call it I. We first observe for a smaller case: Let $n = 3$, $d = 5$, $i = 1$ and $j = 4$. When we call HANOI with the inputs $(1, 4, 3)$, since $n \neq 1$, the **IF** statement is **FALSE** and moves to the **ELSE** condition. Which calls HANOI$(1, 0, 2)$ because

$$[(1+4)2^{-1}] \bmod 5 = [(5)(3)] \bmod 5 = 0$$

But this calls HANOI$(1, 3, 1)$ because

$$[(1+0)2^{-1}] \bmod 5 = [(1)(3)] \bmod 5 = 3$$

After this the **IF** statement is satisfied, since $n = 1$, and the algorithm moves disk 1 from Tower 1 to Tower 3.

From observing this simpler case we can generalize the increment at which disk 1 is moving. Let $\beta = [(i+j)2^{-1}] \bmod d$ then

$$I = ...[[[[i+\beta]2^{-1}+i]2^{-1}]+i]2^{-1}... \bmod d$$

Notice this can get messy very quickly. But if we let $i = 0$ this easily cleans it up and gives us $I = [(j)(2^{-1})^{n-1}] \bmod d$. After analysing the value for I we come to the iterative algorithm with $i = 0$:

MOVE SMALLEST DISK $[(j)(2^{-1})^{n-1}] \bmod d$ TOWER(S) CLOCKWISE
    WHILE A DISK, OTHER THAN SMALLEST, IS ABLE TO MOVE
        DO MOVE THAT DISK
        MOVE THE SMALLEST DISK $[(j)(2^{-1})^{n-1}] \bmod d$
        TOWER(S) CLOCKWISE
    ENDWHILE

The proof of proposition 3.4 is similar to the proof of the count algorithm in the section 3.5.

Proposition 3.4. *The iterative algorithm HANOI ITERATIVE correctly moves n disks from Tower* 0 *to Tower j.*

3.3.3. *Count Algorithm.* From the iterative algorithm it is clear that every other move involves moving disk 1. This will help define the count algorithm which involves using the counter to determine which disk should be moved. To better understand this we examine, in Table 1, the sequence of steps involved with solving the puzzle for 5 disks and 5 towers.

T_0	T_1	T_2	T_3	T_4	Dec. Count	Binary Count	DISK	FROM	TO
12345	-	-	-	-	0	00000	1	0	4
2345	-	-	-	1	1	00001	2	0	3
345	-	-	2	1	2	00010	1	4	3
345	-	-	12	-	3	00011	3	0	1
45	3	-	12	-	4	00100	1	3	2
45	3	1	2	-	5	00101	2	3	1
45	23	1	-	-	6	00110	1	2	1
45	123	-	-	-	7	00111	4	0	2
5	123	4	-	-	8	01000	1	1	0
15	23	4	-	-	9	01001	2	1	4
15	3	4	-	2	10	01010	1	0	4
5	3	4	-	12	11	01011	3	1	2
5	-	34	-	12	12	01100	1	4	3
5	-	34	1	2	13	01101	2	4	2
5	-	234	1	-	14	01110	1	3	2
5	-	1234	-	-	15	01111	5	0	4
-	-	1234	-	5	16	10000	1	2	1
-	1	234	-	5	17	10001	2	2	0
2	1	34	-	5	18	10010	1	1	0
12	-	34	-	5	19	10011	3	2	3
12	-	4	3	5	20	10100	1	0	4
2	-	4	3	15	21	10101	2	0	3
-	-	4	23	15	22	10110	1	4	3
-	-	4	123	5	23	10111	4	2	4
-	-	-	123	45	24	11000	1	3	2
-	-	1	23	45	25	11001	2	3	1
-	2	1	3	45	26	11010	1	2	1
-	12	-	3	45	27	11011	3	3	4
-	12	-	-	345	28	11100	1	1	0
1	2	-	-	345	29	11101	2	1	4
1	-	-	-	2345	30	11110	1	0	4
-	-	-	-	12345	31	11111	-	-	-

Table 1: Towers of Hanoi Solution for 5 disks and 5 towers

Observe that the rightmost 0 in the binary count determines which disk is to move. Let us label the rightmost bit to be in position 1 and the next rightmost bit to be in position 2 and so on. This is easily seen from Table 1 and is in fact similar to the standard count algorithm done by Cull [10]. In the standard count algorithm the even disks always move in the

opposite direction as disk 1, while the odd disks move in the same direction as disk 1. For d TOWERS and n DISKS: Define function F as $F(x) = [(j)(2^{-1})^{x-1}] \bmod d$. Then disk 1 will always move $F(n)$ towers clockwise. Disk 2 will always move $F(n-1)$ towers clockwise. Disk 3 will always move $F(n-2)$ towers clockwise and so on until disk n, which will move $F(n-(n-1)) = F(1) = (-1) \bmod d$. To sum this all up disk δ will move $[(j)(2^{-1})^{n-\delta}] \bmod d$ clockwise. From this information we construct a general table for solutions of Towers of Hanoi.

Binary Count	DISK	FROM	TO
0...0000	1	0	$(-1)(2^{-1})^{n-1} \bmod d$
0...0001	2	0	$(-1)(2^{-1})^{n-2} \bmod d$
0...0010	1	$(-1)(2^{-1})^{n-1} \bmod d$	$(-2)(2^{-1})^{n-1} \bmod d$
0...0011	3	0	$(-1)(2^{-1})^{n-3} \bmod d$
0...0100	1	$(-2)(2^{-1})^{n-1} \bmod d$	$(-3)(2^{-1})^{n-1} \bmod d$
0...0101	2	$(-1)(2^{-1})^{n-2} \bmod d$	$(-2)(2^{-1})^{n-2} \bmod d$
0...0110	1	$(-3)(2^{-1})^{n-1} \bmod d$	$(-4)(2^{-1})^{n-1} \bmod d$
0...0111	4	0	$(-1)(2^{-1})^{n-4} \bmod d$
0...1000	1	$(-4)(2^{-1})^{n-1} \bmod d$	$(-5)(2^{-1})^{n-1} \bmod d$
0...1001	2	$(-2)(2^{-1})^{n-2} \bmod d$	$(-3)(2^{-1})^{n-2} \bmod d$
0...1010	1	$(-5)(2^{-1})^{n-1} \bmod d$	$(-6)(2^{-1})^{n-1} \bmod d$
0...1011	3	$(-1)(2^{-1})^{n-3} \bmod d$	$(-2)(2^{-1})^{n-3} \bmod d$

Table 2: Towers of Hanoi Solution that will move n disks from Tower 0 to Tower $d-1$, where there are d towers.

After observing the properties from Table 1 and Table 2 we generalize the count algorithm that would solve moving n disks from Tower 0 to Tower $d-1$, for d towers:

PROCEDURE TOWERS(n)
T:= 0 (*TOWER NUMBER COMPUTED MODULO d*)
COUNT:= 0 (*COUNT HAS n BITS*)

P:= $[(-1)(2^{-1})^{n-1}] \bmod d$

WHILE TRUE DO
MOVE DISK 1 FROM T TO T+P
T:= T+P
COUNT:= COUNT + 1
IF COUNT = ALL 1's THEN RETURN
IF RIGHTMOST 0 IN COUNT IS IN POSITION b
THEN MOVE DISK b FROM T+ $[(2^{b-2})(2^{-1})^{n-1}] \bmod d$
to T-$[(2^{b-2})(2^{-1})^{n-1}] \bmod d$
COUNT:= COUNT + 1
ENDWHILE

In order to prove the upcoming proposition we will show when the **COUNT=** 2^k-1 the count algorithm has completed the same moves as HANOI$(0,[-(2^{k-2})(2^{-1})^{n-1}] \bmod d,k)$.

Lemma 3.5. *When decimal count* $=2^k-1$*, that is binary* ***COUNT*** $=00...01...1$ *with* $k1's$*, then the correct moves for HANOI*$(0,[-(2^{k-1})(2^{-1})^{n-1}] \bmod d,k)$ *have been completed by the count algorithm and disk* 1 *is on tower* $[-(2^{k-1})(2^{-1})^{n-1}] \bmod d$.

Proof. BASE CASE: If $k=1$, **COUNT** $=0...01$, the single move T to $T+P$ has been completed, where $T=0$ and $T+P=[(-1)(2^{-1})^{n-1}] \bmod d$. Because$[-(2^{k-1})(2^{-1})^{n-1}]=[(-1)(2^{-1})^{n-1}]$ for when $k=1$ this completes the moves for HANOI $(0,[(-1)(2^{-1})^{n-1}] \bmod d,1)$. In addition, disk 1 is on tower $[(-1)(2^{-1})^{n-1}] \bmod d$. This agrees with our claim.

INDUCTIVE STEP: Observe that the **COUNT** can only equal the value 2^k-1 immediately before the **IF....RETURN** statement. Assume the moves for HANOI$(0,[-(2^{k-1})(2^{-1})^{n-1}] \bmod d,k)$ have been completed and disk 1 is on tower $[-(2^{k-1})(2^{-1})^{n-1}] \bmod d$. We want to show when **COUNT** $=2^{k+1}-1$ the moves for HANOI$(0,[-(2^k)(2^{-1})^{n-1}] \bmod d,k+1)$ have been completed and disk 1 is on tower $[-(2^k)(2^{-1})^{n-1}] \bmod d$.

The next move would involve knowing where the rightmost 0 is within the **COUNT**. This is very simple since the rightmost 0 in the **COUNT** would be in position $k+1$. This would move disk $k+1$ from tower 0 to tower

$$-2[(2^{k-1})(2^{-1})^{n-1}]=[-(2^k)(2^{-1})^{n-1}].$$

Next **COUNT** will be incremented to 0...010...0, where there are $k0's$ after the 1. And when the **COUNT** $= (2^{k+1}-1$, the algorithm will have repeated the same sequence of moves as before since it only *sees* the rightmost information in **COUNT**, with the difference that T will have started with a different value. Therefore the next moves up until **COUNT** $= (2^{k+1}-1$ would have moved the k disks from tower $[-(2^{k-1})(2^{-1})^{n-1}] \bmod d$ to tower $[-(2^k)(2^{-1})^{n-1}]$. At this point it is clear that we have moved $k+1$ disks from tower 0 to tower $[-(2^k)(2^{-1})^{n-1}]$ and disk 1 is on tower $[-(2^k)(2^{-1})^{n-1}]$. We conclude, by induction, when **COUNT** $= 2^k-1$ the count algorithm has made the same moves as HANOI$(0,[-(2^{k-1})(2^{-1})^{n-1}] \bmod d,k)$. □

Proposition 3.6. *The count algorithm TOWERS*(n) *correctly moves n disks from tower* 0 *to tower j.*

Proof. Lemma 3.5 tells us that TOWERS(n) applies the same moves as HANOI when **COUNT** $= 2^k-2$. Since HANOI has been proven to be correct we know TOWERS is also correct as long as the last **COUNT** is in the form 2^k-1, which it is since the total number of moves is 2^n-1. □

4. Dimension 2^m Labeling and Puzzle

Previous work has also established labels and puzzles for dimension 2^m. [3] The labeling on dimension 2^m has been established to have finite-state machines for codeword recognition and error correction. This labeling also has the Gray code property and corresponds to the dimension 2^m puzzle. In the case $d = 2$ the dimension 2^m labeling corresponds to the Spin-Out puzzle by ThinkFun.

4.1. **The Spin-Out Puzzle.** The graphs and labelings for dimension 2^m iterated complete graphs are based on the Spin-Out puzzle. The goal of the game is to remove a rectangle with seven spinners on it from a plastic case. In the traditional starting position, all seven spinners are vertical, and the rectangle can only be removed when all of the spinners are aligned horizontally. Let the spinners be labeled 0 to 6 from the left to the right. The n^{th} spinner can only be turned when the spinners 0 through $n-2$ are horizontal and spinner $n-1$ is vertical. Note that the leftmost spinner is free to move at anytime.

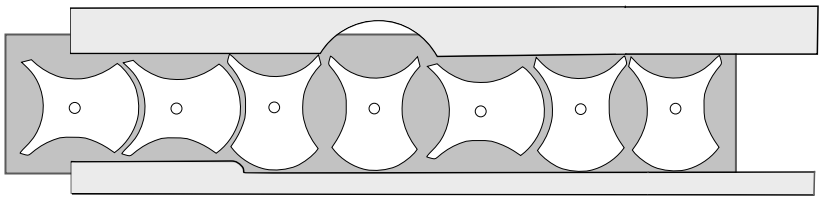

FIGURE 10. A configuration of the Spin Out puzzle, which corresponds to the labeling 0011011. The spinner under the arc may move, and we may also slide the large rectangle to the right and move the the leftmost spinner.

To represent this puzzle by a labeling on a graph, let each spinner be represented by a bit. If the spinner is horizontal, the bit is 0; if it is vertical, 1. Then let each configuration of the puzzle be represented by a string of seven bits, the leftmost bit corresponding to the leftmost spinner and so on. We associate these labels with vertices, and when we create edges between them representing possible moves of the Spin-Out puzzle, we get a Gray labeling on K_2^7.

Note that the puzzle can be generalized to use any number of spinners, not just 7. This resulting family of puzzles can be represented by the reflected binary Gray code on K_2^n. Figure 23 shows the graphs of K_2^1, K_2^2, and K_2^3. An easily defined recursive construction for dimension 2 graphs is presented by Savage.

4.2. **The Dimension 2^m Puzzle.** Previous work shows there exists an easy extension of Spin-Out to all dimensions which are powers of 2. The extended puzzles will retain the sliding aspect of Spin-Out, but the spinners will be replaced by pieces which consist of a stack of spinners. When a piece is composed of m spinners, it will have 2^m possible orientations, since each spinner can be in one of two orientations. For n pieces, there will be $(2^m)^n = d^n$ configurations. The sliding rules will determine which pieces can change, and new spinning rules will determine how the pieces can change. Together they will define which configurations can change to which configurations.

Stubak and Stevenson [1] provided a way to associate orientations of the pieces with numbers 0 through $d-1$ by defining the orientations of the pieces as follows:

- For a dimension $d = 2^m$, each puzzle piece will consist of m spinners stacked one on top of the other.
- To find the orientation of piece j, write j as a binary number. To set a piece in this orientation, let the 1's (rightmost) bit represent the top spinner; a 0 bit means that it is horizontal, while a 1 bit means that it is vertical. Similarly, let the 2's bit represent the spinner just below the top spinner, the 4's bit the next spinner, etc. Continue in this manner; the $(m-1)$'s bit will represent the bottom spinner.
- Thus for each $j \in \{0,\ldots,d-1\}$ there is a distinct orientation and corresponding binary number.

Example 4.1. *Suppose $d = 8 = 2^3$. That is, $m = 3$, so the pieces are composed of 3 spinners. Then, for example, the $0 = 000_2$ orientation consists of all horizontal spinners, the $7 = 111_2$ orientation has all vertical spinners, and the $3 = 011_2$ orientation has a horizontal spinner on the bottom with two vertical spinners above it.*

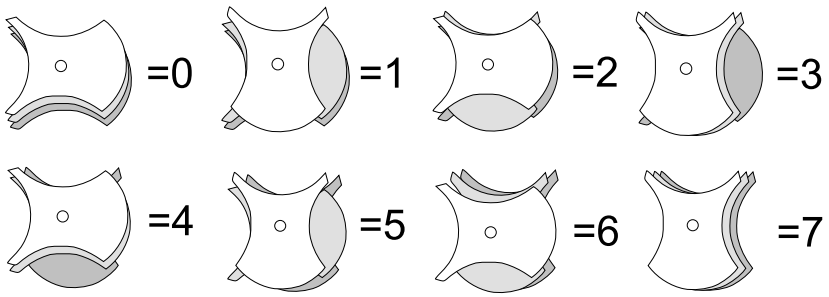

FIGURE 11. Piece orientations for the Dimension 8 Puzzle

Note that the 0^{th} orientation will always consist of all horizontal spinners.

For an iteration n for $n \geq 1$, there will be n puzzle pieces. Call the leftmost piece the 0^{th} piece and continue numbering the pieces from left to right. Thus the rightmost piece is the $(n-1)^{\text{st}}$ piece. Given a configuration of the puzzle, there is a labeling with a string of characters from $\{0,\ldots,d-1\}$, where each piece 0 through $n-1$ is represented by the number of the orientation it is in. $f(j)$ refers to the orientation of piece j.

Example 4.2. *Continuing from the example above, Figure 12 has the label 0374.*

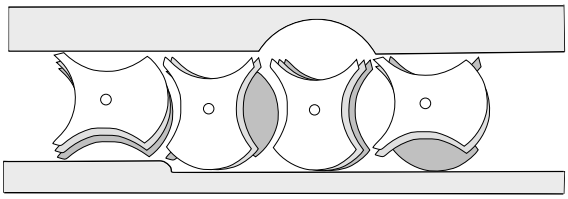

FIGURE 12. An example configuration for the Dimension 8 Puzzle. The pieces are numbered left to right 0, 1, 2, and 3.

The rules of this puzzle are an extension of the rules of the Spin-Out Puzzle.

(1) The 0^{th} piece may always change orientation, and may change to any other orientation.
(2) To spin at least one spinner of the j^{th} piece, $f(0)$ through $f(j-2)$ must be 0 and $f(j-1) \neq 0$; that is, pieces 0 through $j-2$ are of orientation 0 and piece $j-1$ is not orientation 0. If these conditions are satisfied, then move as many spinners of the j^{th} piece as possible; that is, any spinner that can switch between its horizontal and vertical positions must do so.
(3) The goal of the puzzle is, given some initial configuration, to move all the pieces to orientation 0.

Example 4.3. *In Figure 12, piece* 2 *is able to change orientations. Since the bottom spinner of piece* 1 *is horizontal, the bottom spinner of piece* 2 *cannot move. However, the other two spinners can move, and so they must become horizontal. Thus the orientation of piece* 2 *must change from* 7 *to* 4.

4.3. **Recursive Construction of the Dimension 2^m Labeling.** The recursive labeling of graphs of dimension 2^m as presented by Stubak and Stevensen [1] is as follows:

(1) Label some "top" vertex 0 and label in order counterclockwise, and call this labeling L_d^1.

(2) L_d^n is based on d copies of L_d^{n-1}. In order to neatly depict the graph, it is neccessary to permute each subgraph so that the edges are in the desired locations. To create L_d^n:

- When $i = 0$, the copy is placed in the top (0^{th}) position and 0 is appended.
- For all other i, the permutation Γ_i is applied to the last character of each label in the i^{th} copy of L_d^{n-1}, where Γ_i bitwise adds i to the last character in a label. That is, $\Gamma_i(\ldots x) = \ldots(x \oplus i)$. Then this i^{th} copy is placed in the i^{th} position counterclockwise from the top position, and the character i is appended to each label.
- Finally, for each i, the vertex at position j from the top position is connected to the i^{th} corner of j^{th} subgraph. If $j = i$, the vertex is a corner of the entire graph and no edge is drawn.

The operator $\oplus$ denotes bitwise addition on two numbers; that is, $r \oplus s$ means write both r and s as binary numbers and do a bitwise addition (also known as a XOR, or addition without carry).

It is important to note that unfortunately this recursive labeling of the family of iterated complete graphs of these dimensions is not unique. There exist other labelings that still preserve the desired properties.

Example 4.4. *Figure 13 shows L_8^1 and L_8^2, the recursive labeling for the graphs K_8^1 and K_8^2. As an example of how to permute a subgraph, look at the subgraph immediately counterclockwise of the top position, the 1^{st} subgraph. This was labeled by applying Γ_1 to L_8^1 and appending 1. Note that $0 \oplus 1 = 1$, $1 \oplus 1 = 0$, $2 \oplus 1 = 3$, etc.*

4.4. **Algorithms to Solve the Dimension 2^m Puzzle.** In the Spin-Out puzzle there are n spinners and 2 possible orientations for each spinner. In the Dimension 2^m puzzle there are n stacks of spinners and d possible orientations for each stack. Thus the Dimension 2^m puzzle is simply an extension of the Spin-Out puzzle. An algorithm to solve the Dimension 2^m puzzle will therefore be similar to an algorithm to solve the Spin-Out puzzle.

4.4.1. *Recursive Algorithm.* Pruhs [11] presents a two part recursive algorithm to solve the Spin-Out puzzle, which consists of moving the spinners from 11...1 to 00...0, with n spinners. A similar recursive algorithm can be written to solve the Dimension 2^m puzzle with n stacks of spinners and d possible orientations. To solve the puzzle move the stacks of spinners from $(d-1)(d-1)...(d-1)$ to 00...0. Let $rotate(i)$ mean to rotate spinner number i from $d-1$ to 0 or from 0 to $d-1$, where $i \in \{0,...,n-1\}$, and the stacks of spinners are indexed from 0 to $n-1$ from left to right.

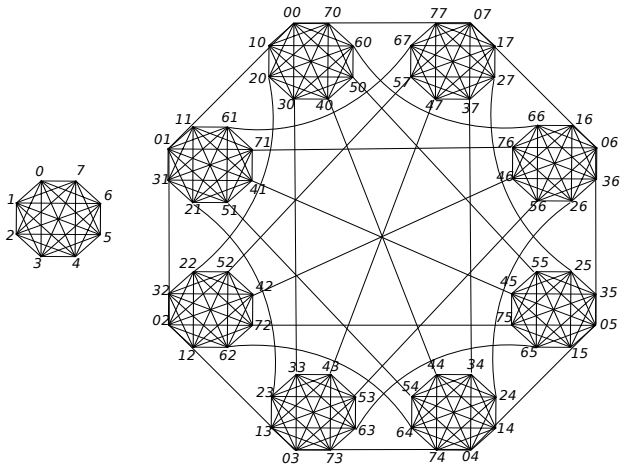

FIGURE 13. The labeling for the first and second iterations for the dimension 8 graph, corresponding to the extended Spin-Out puzzles with 1 and 2 pieces respectively.

PROCEDURE A(n:Integer)
Comment: takes puzzle from $(d-1)^n$ to $0^{n-2}(d-1)0$
IF $n = 1$ THEN $rotate(0)$
ELSE IF $n = 2$ THEN $rotate(1)rotate(0)$
ELSE BEGIN
A($n-2$)
$rotate$(n)
C($n-1$)
END
END

PROCEDURE C(k:Integer)
Comment: takes puzzle from $0^{k-1}(d-1)$ to 0^k or from 0^k to $0^{k-1}(d-1)$
IF $k = 1$, THEN $rotate(0)$
ELSE IF $k = 2$ THEN $rotate(0)rotate(1)$
ELSE BEGIN
C($k-1$)
$rotate$(k)
C($k-1$)
END
END

Proposition 4.5. *The recursive algorithm correctly solves the* 2^m *puzzle.*

The proof of proposition 4.5 is very similar to the proof of the recursive algorithm for the Spin-Out puzzle presented by Pruhs [11].

4.4.2. *Iterative Algorithm.* There also exists an iterative algorithm to solve the 2^m puzzle, which produces the same sequence of rotations as the recursive algorithm. This algorithm is also similar to the iterative algorithm that solves the Spin-Out puzzle presented by Pruhs [11].

IF n is odd
    THEN ROTATE stack 0 from 0 to $d-1$
        WHILE A stack other than stack 0 can rotate
            Do ROTATE that stack
            ROTATE stack 0 from 0 to $d-1$ or from $d-1$ to 0
        ENDWHILE

5. Puzzle and Labeling for Other Even Dimensions

In addition to the labelings corresponding to puzzles on the families of iterated complete graphs of odd dimensions and of dimensions that are powers of two, previous work has established puzzles and labelings corresponding to other even dimensions. [1] The SF Puzzle is a Towers-of-Hanoi-like puzzle, which corresponds to odd-dimensional graphs, and the Dimension 2^m puzzle, based on the Spin-Out puzzle, corresponds to graphs of dimension 2^m. These two puzzles are completely different but they can be combined to produce the puzzle and the labeling that corresponds to graphs of other even dimensions.

5.1. **The Combination Puzzle.** Every even number can be written as $q \cdot 2^m$ where q is odd and $m \geq 1$. Thus, it is possible to combine the two types of puzzles, an SF puzzle of dimension q and an extended Spin-Out puzzle with dimensiong 2^m, to define a general puzzle for any dimension. The goal of these puzzles is a combination of the SF and Spin-Out goals. That is, given some initial configuration, to move all the pieces to orientation 0 on a specific tower.

As in the SF puzzle, there are n pieces stacked on q towers labeled $0, ..., q-1$, from left to right. In the SF puzzle, the pieces are disks that have no orientation, so that only a piece's tower matters. But now, there are n pieces consisting of m spinners, and each piece has 2^m possible orientations. The possible orientations are numbered 0 through $2^m - 1$, on each of q towers. These orientations are defined exactly as in the extended Spin-Out puzzle, by writing the orientation in binary and letting each bit represent one spinner of the piece. For a given piece j, combine the tower t_j and orientation r_j to define its **total orientation** $f(j)$ as $t_j \cdot 2^m + r_j$. Therefore each piece has $q \cdot 2^m = d$ possible total orientations in all. This makes d^n configurations for the puzzle with n pieces.

Possible orientations for the Dimension 6 puzzle.

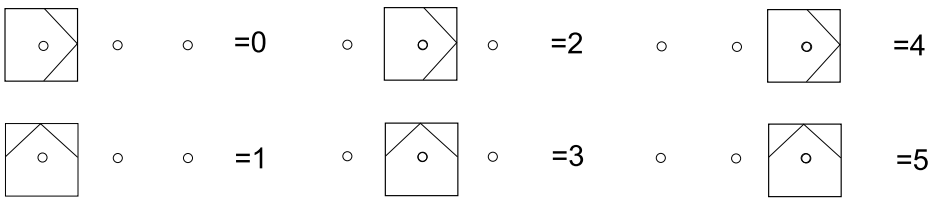

FIGURE 14. Piece positions for the Dimension 6 Puzzle. The smaller example of the combined puzzle where $d = 3 \cdot 2^1$

Pieces are numbered from the 0^{th} piece, the least restricted (which can be thought of as the smallest or leftmost piece), through the $(n-1)^{\text{st}}$ piece, the most restricted (biggest or rightmost). The configurations of the puzzle are labeled by strings representing the orientations of the pieces. The leftmost digit will correspond to the smallest piece and continue in order of size with the rightmost digit corresponding to the largest piece. For example, the game configuration for the Dimension 6 puzzle, iteration $n = 2$, in Figure 15 corresponds to the label 50.

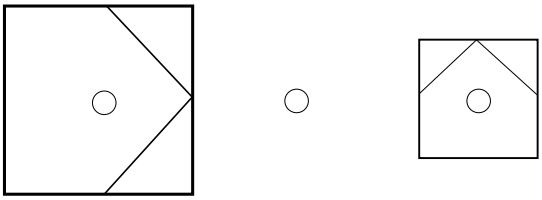

FIGURE 15. An Example Configuration of the Dimension 6 Puzzle

5.2. **Rules of the Combination Puzzle.** The original Towers of Hanoi rules still apply to the Combination puzzle. Pieces must always be stacked from largest on the bottom to smallest at the top. Only the smallest piece on a tower may be moved, and it may only move to a tower containing either no pieces or only pieces larger than itself. As usual, the smallest piece may always move in any way, to any orientation. There are three rules that define legal moves between configurations:

(1) **The 0^{th} Piece Rule** The 0^{th} piece may always move to any other total orientation.
(2) **Conditions for Movement** For any $j \neq 0$, the j^{th} piece may move if all of the following conditions are true.
   (a) the total orientations of piece 0 through $j-2$ are all the same and are equivalent to 0 mod 2^m; that is, they are on the same tower and their orientations are 0
   (b) t_{j-1} is the same as t_0 through t_{j-2}; i.e., pieces 0 through $j-1$ are all on the same tower
   (c) if $t_j = t_{j-1}$, then $f(j-1)$ is not the same as $f(0)$ through $f(j-2)$; that is, if all pieces 0 through j are on the same tower, then piece $j-1$ has $r_{j-1} \neq 0$
(3) **The Total Orientation Change Function** If the Conditions for Movement are satisfied, the tower of piece j may change to

$$(2t_{j-1} - t_j) \text{mod q}$$

at the same time as its orientation changes to

$$[f(j-1) \oplus f(j)] \bmod 2^{\mathrm{m}} \; = \; \mathrm{r_{j-1}} \oplus \mathrm{r_j}$$

Note that, conditions (a) and (c) are exactly the dimension 2^m conditions, and that condition (b) is exactly the SF Puzzle condition. Also, as expected, if $q = 1$ this definition reduces to the Dimension 2^m puzzle, and if $m = 0$ it reduces to the SF Puzzle.

A few examples of moves can be found in Figure 16.

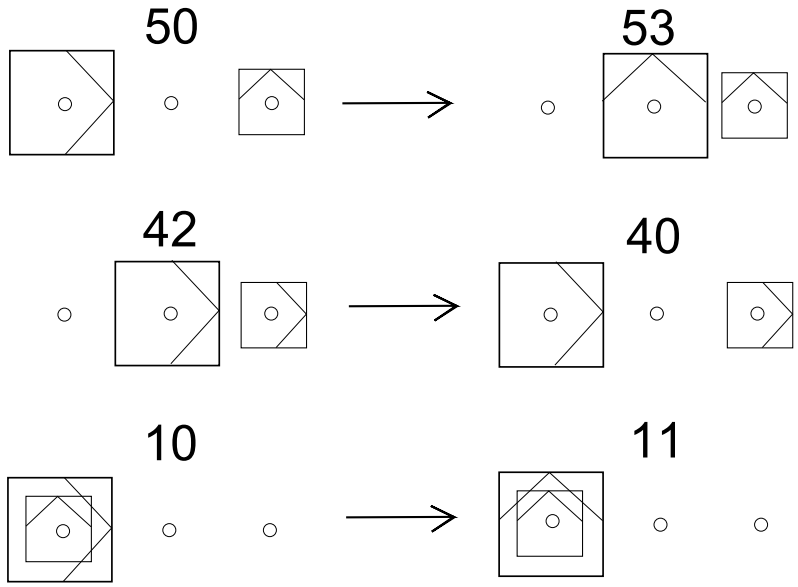

FIGURE 16. Example moves for the Puzzle on Dimension 6

5.3. **The Recursive Construction of General Dimensions Labeling.** These rules give us the labeling of the graph. Figure 17 shows the labeling of the Dimension 6 graph. Note that there are two Towers of Hanoi labelings embeded in each graph, and that there are three reflected binary Gray code labelings on three of the outside edges (each with some simple perumutations of characters).

As seen by section 5, Skubak and Stevenson [1] were able to describe the rules for the generalized dimension puzzle and create a labeling from these rules. However, they were not able to define a recursive construction for labeling these graphs. The first attempt to create such a construction was to use previous construction and possibly combine them. Notice, from the labeling of figure 17, the 0th copy has a unique permutation that is different from the SF labeling and the dimension 2^m labeling. For the 0th copy the leftmost bit is organized in pairs, namely: $(0,1),(2,3),(4,5)$. They are

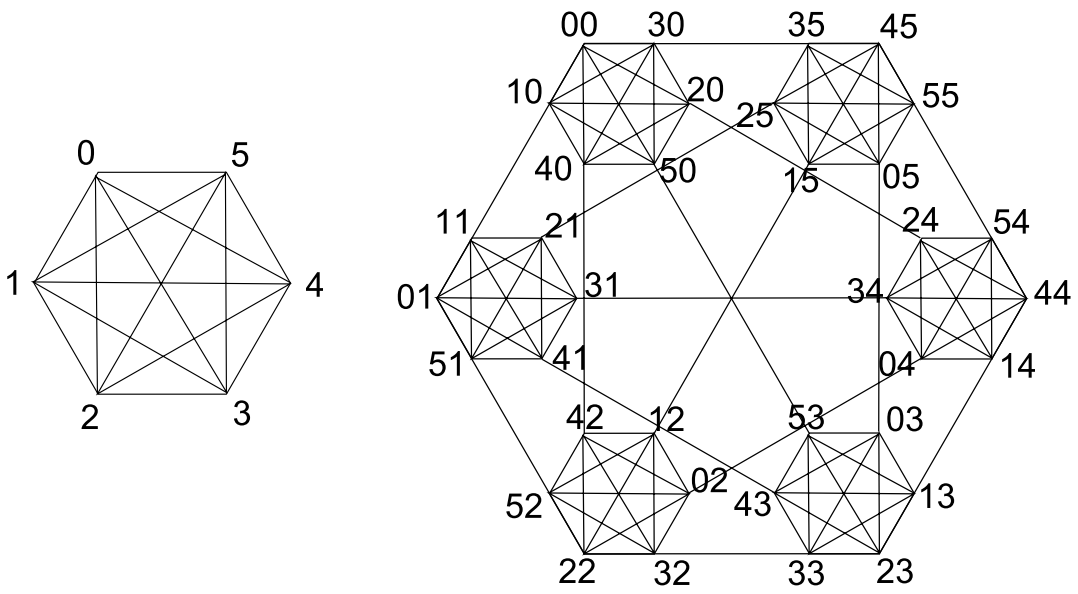

FIGURE 17. The Puzzle Labling for K_6^1 and K_6^2

then labeled as: $(0,1,4,5,2,3)$ in counter clockwise order. We see this property extends to further dimensions, such as 10 and 12. Dimension 10 groups as: $(0,1),(2,3),(4,5),(6,7),(8,9)$. And dimension 12 groups as: $(0,1,2,3),(4,5,6,7),(8,9,10,11)$. They are then labeled by counting every (-2^m) mod q group and organizing them counter clockwise. This is easily seen in figure 18 and figure 19. The hope is this can be generalized to help create the 0th copy. But further investigation must be done to show it extends to all dimensions $q \cdot 2^m$.

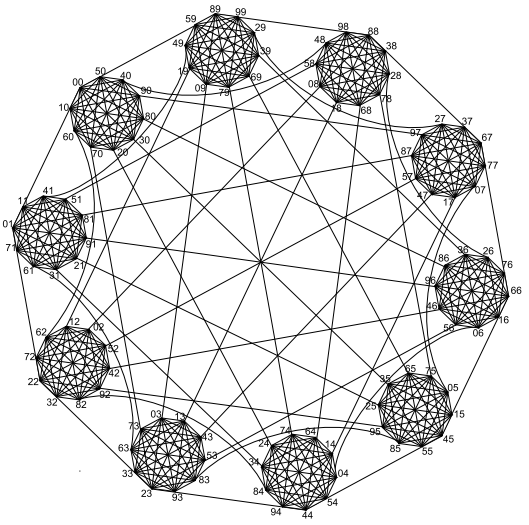

FIGURE 18. The Puzzle Labeling for K^2_{10}

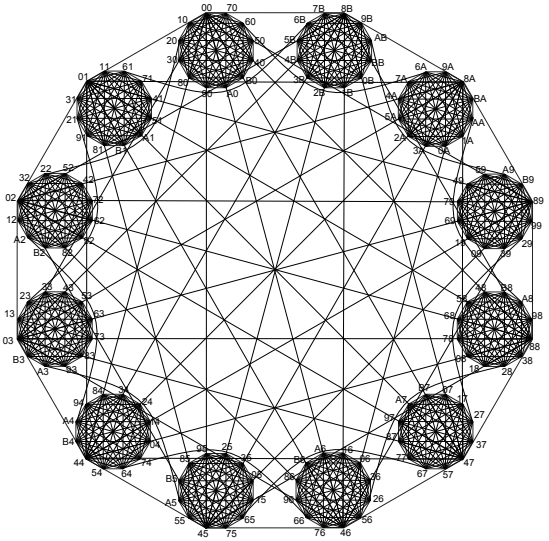

FIGURE 19. The Puzzle Labeling for K^2_{12}, notice in the labeling that ten is represented by an *A* and eleven is represented by a *B*.

6. ENCODING AND DECODING

As stated in section 2.3, an encoding and decoding scheme for a particular labeling of K^n_d is a bijection between the integers and the set of codewords in the labeling. This section discusses previous attempts at an encoding and decoding scheme on labelings of the iterated complete graphs

as well as attempts to simplify and generalize encoding and decoding for these families of graphs.

6.1. **Previous Findings.** Previous work has been done on encoding and decoding for these iterated complete graphs. Cull and Nelson [7] provided an easy encoding and decoding for the Towers of Hanoi where $d = 3$. It uses the fact that each distance 1 neighborhood of a codeword contains exactly one vertex whose label is a multiple of four when the label is read as a base 3 number. They denote by G_n the set of codewords in the SF labeling of K_3^n.

To encode and decode for K_3^n:

- CODE is given by:

$$CODE(I) = ERROR - CORRECT(4 * I)$$

- DECODE is given by:

$$DECODE(x) = N(x)/4$$

  where $N(x)$ is the unique number divisible by four which is associated with a node in the neighborhood of x.

Russel [13] showed that Cull and Nelson's technique could not be easily generalized for $d > 3$ and odd. In the general case, some codewords are not adjacent to any multiple of $d+1$, while others are adjacent to two multiples of $d+1$. The possibility of extension to even dimensions had not yet been explored.

Russel [13] also provides an algorithm for an encoding and decoding scheme for the SF labeling with arbitrary d and n. However, this scheme is complex and uses few of the theoretical properties of the SF labeling. The hope was to create a simple method for encoding and decoding for all K_d^n using the labelings of the graphs.

6.2. **Base b division by $b+1$.** After viewing previous work done with encoding and decoding, on the standard Towers of Hanoi, the aim was to generalize for all dimensions, if possible. As a tool, we wanted a simple method for dividing a base b number by $b+1$. Of course, we wanted our method to provide both the quotient and the remainder. Here we introduce the finite state machines that will be used to compute the division.

Definition 6.1. *Let $A = \{a, ...\}$ be a finite alphabet. A **finite automaton** over A consists of the following items:*

(1) *a finite nonempty set F, called the set of **states**;*
(2) *a subset D of F, called the set of **final states**;*
(3) *a distinguished element $s_0 \in F$, called the **start state**;*
(4) *unary functions $f_x : F \to F$ one for each $x \in A$, $i = 1, ..., m$, called the **transition functions**.*

This automation recognizes the string s, when the automation starts in the start state follows the transitions for the consecutive characters of s and ends in a final state. If the automation does not recognize s, the automation is said to reject s.

Definition 6.2. *A **finite transducer** is like a finite automaton, but each transition has an associated output symbol. So, the transducer produces an output string for each input string. As in the automaton, the transducer starts in the start state and processes the input string one character at a time.*

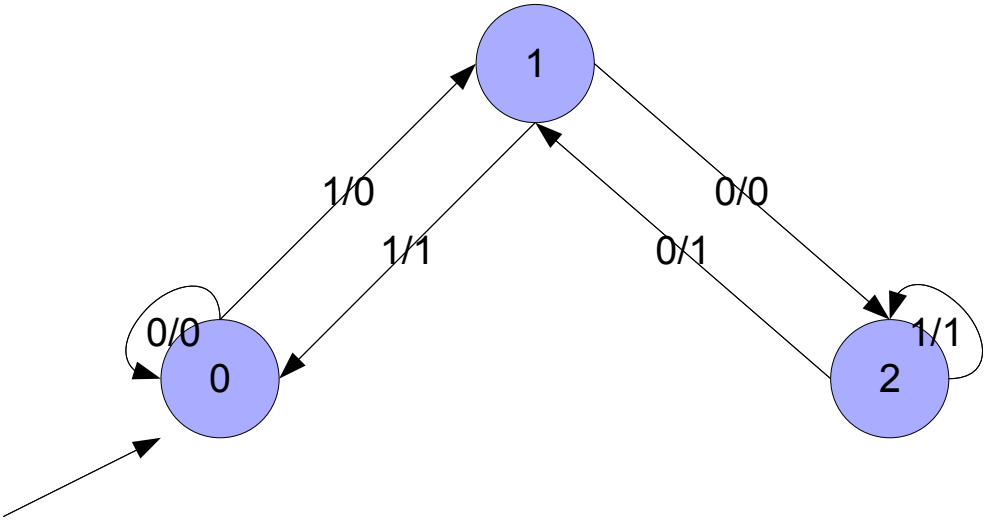

FIGURE 20. Finite State Transducer (FST) for Base 2 Division by 3. Given a binary number, begin at the 0 state, and feed the number in from high order bit to low order bit. The machine outputs an integer in binary which corresponds to the correct quotient, and ends at the state corresponding to the remainder. For example, if the input is 111, the machine outputs 010 and ends at state 1. That is, 7 divided by 3 equals 2 remainder 1. This is the same process used for the general FST which performs base b division by $b+1$.

The finite state transducer shown in figure 20 correctly divides any binary number by 3, as each bit is fed in, the machine outputs the corresponding bit of the quotient, and ends at state r, the remainder. Figure 21 shows the FST that divides any ternary number by 4 and ends at state r. Figure 22

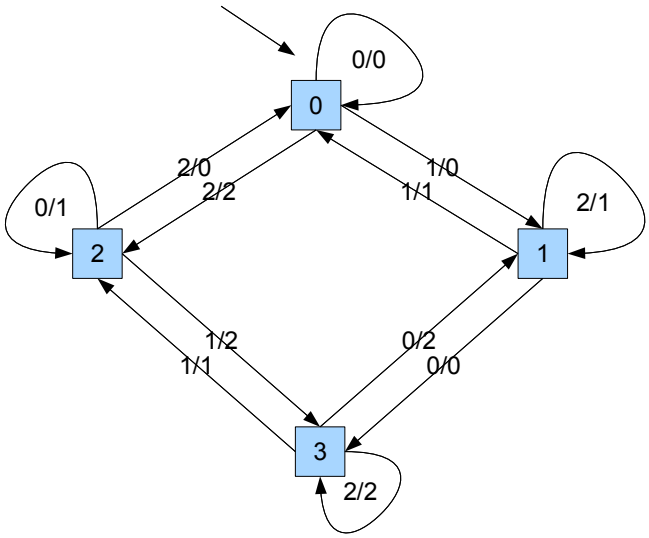

FIGURE 21. FST for Base 3 Division by 4

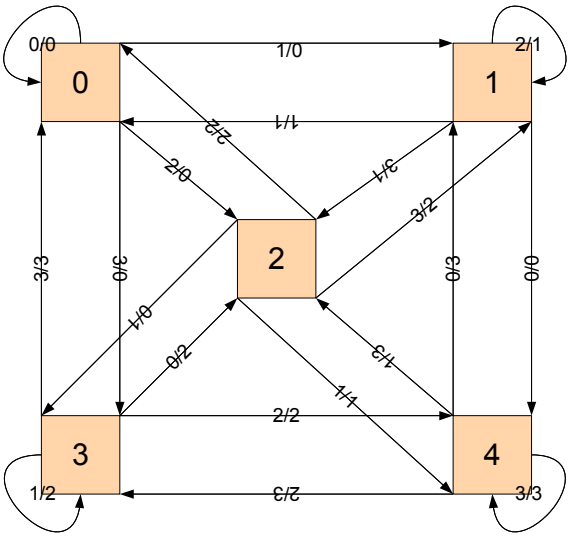

FIGURE 22. FST for Base 4 Division by 5

shows the FST that divides any number in base 4 by 5 and ends at state r. We will now generalize the FST for division in base b numbers by $b+1$.

To construct the FST that takes an string in base b and outputs the string divided by $b+1$, in base b, we will need $b+1$ states from the set $\{0,1,2....,b\}$ to represent the possible remainders. State 0 will be the starting state and

will be adjacent to all $a \in \{0,1,2,....,b-1\}$. Moreover, each directed edge that goes from state 0 to state a inputs a and outputs 0. Now, let x be a string with more than 1 digit. Assume the input ends at state r. Then there is an edge between state r and $d-r$ for all $d \geq r$ where the input is d and output is r. And for all $d < r$ there is an edge between state r and $b+d-(r-1)$ where the input is d and output is $r-1$. The proof of Theorem 1 will explain the reasoning behind the construction of this FST.

Theorem 6.3. *The finite-state transducer, described above, correctly divides x, in base b, by* $b+1$ *and ends at state r, the remainder in the division.*

Proof. Let Q be the quotient and r be remainder when x is divided by $b+1$ in base b. BASE CASE: Let x be of length $n=1$. Then $x \in \{0,1,2,...,b-1\}$, which implies $x < b+1$. Thus when we divide x by $b+1$ we will get $Q=0$ and $r=x$. This satisfies our description of the FST since every state adjacent to 0, the starting state, is an element from the set $\{0,1,2,...,b-1\}$. Moreover, each edge going from 0 to x inputs x and outputs 0, the quotient, and goes to the correct remainder state.

INDUCTIVE STEP: Now Assume the FST works for $n=k$ bits. Then we know $x=(b+1)Q+r$. We will show the FST works for $n=k+1$ bits. By appending d to x we get two cases, $d \geq r$ and $d < r$. In each case the number x can be replaced by $\hat{x} = b \times x + d$. This should be clear using arithmetic in base b. And we can easily get $\hat{x} = b(b+1)Q+br+d$ using substitution. Notice, the first k digits in x would not change since the FST would still go through the same directed edges and would still have the same output for the first k digits. Thus, we observe the output values of d and the ending state, which is the remainder in the division.

Case 1: If $d \geq r$. From the equation $\hat{x} = b(b+1)Q+br+d$ we divide by $b+1$ on both sides to observe the quotient and the remainder. We get $\hat{x} \div (b+1) = bQ + (br+d) \div (b+1)$. By rewriting $(br+d)$ to equal $(br+r+d-r)$ we will cause $\hat{x} \div (b+1) = bQ + r + (d-r) \div (b+1)$ and so $\hat{d} = r$ and $\hat{r} = d-r$. Notice $d-r$ is positive and therefore acceptable as a remainder. Which satisfies out FST.

Case 2: If $d < r$. Similarly we get $\hat{x} = b(b+1)Q+br+d$ and by dividing b+1 we get $\hat{x} \div (b+1) = bQ + (br+d) \div (b+1)$. But we can't have the remainder be $d-r$ since $d-r$ would be negative. Thus we will substitute $(br+d)$ with $[b(r-1)+b+d+(r-1)+(r-1)] = b(r-1)+(r-1)+b+d-(r-1)$. After substitution we get $\hat{x} \div (b+1) = bQ + (br+d) \div (b+1)$ = $bQ+r-1+(b+d-(r-1)) \div (b+1)$. Therefore we get $\hat{d} = (r-1)$ and $\hat{r} = (b+d-(r-1))$. Which satisfies the FST.

Since all cases work correctly we have shown the FST works for $n=k+1$ bits. By induction we can say the FST works for all strings. □

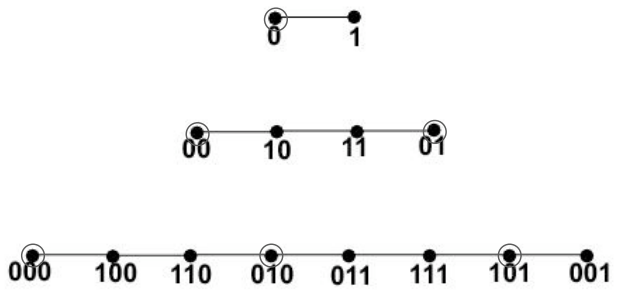

FIGURE 23. The reflected binary Gray code for K_2^1, K_2^2, and K_2^3 with code vertices circled

6.3. **Encoding and Decoding for the Spin-Out Graph.** Using base b divisoin by $b+1$, an encoding and decoding scheme for the graphs of dimension 2 corresponding to the Spin-Out puzzle is simple. Figure 23 demonstrates that if we require that 0...0 is a codeword, the only P1ECC on the graphs K_2^n has the codewords at every third vertex. Thus the encoding and decoding scheme only needs to convert the Gray label to its binary position and divide or multiply by 3, both of which can be simply done with finite-state machines.

To encode and decode for K_2^n:

- CODE is given by:

$$CODE(I) = GRAY(3 * I)$$

- DECODE is given by:

$$DECODE(J) = \lfloor BINARY(J)/3 \rfloor$$

From Theorem 6.3 there exists a finite state machine that divides numbers in base 2 by 3, which is shown in figure 20. There also exists a finite state machine that converts gray to Binary, which is shown in figure 24. Then it is possible to combine these two machines to create a finite state machine which performs DECODE on K_2^n, which is shown in figure 25.

Figure 25 is the finite state machine which performs DECODE for K_2^n. The input for this machine are the labels on K_2^n, that is G_n. The machine reads a string bit by bit, from right to left, following the transition with the given bit on the left side of the label. The machine then changes that bit to the bit on the right side of the label. The output of the machine is an integer which is an element of $\{0,1,...,|G_n|-1\}$, and corresponds to the correct quotient. There are two accepting states in the machine. If the machine ends at one of these accepting states, the label is a codeword. If the machine ends at another state the label is a noncodeword, and that state

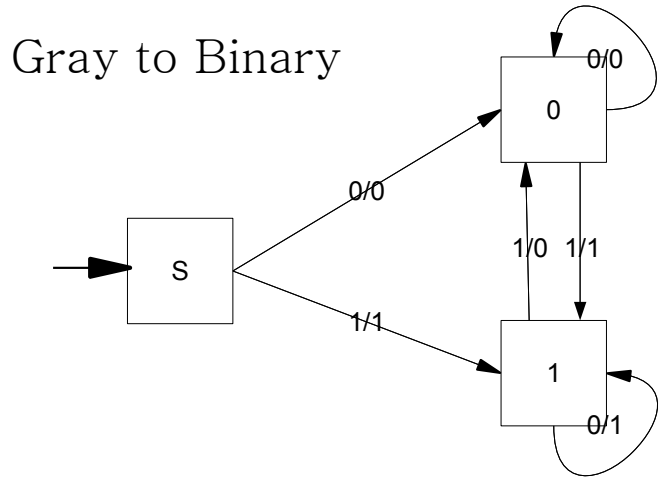

FIGURE 24. Given J in Gray code starting with the high order bit (which corresponds to the rightmost bit of the reflected binary Gray code in the labeling of K_2^n), this machine computes BINARY(J)

corresponds to the remainder. In this case, if the remainder is 1 the output is the nearest integer corresponding to a codeword. If the remainder is 2 the output must be rounded up to find the nearest integer corresponding to a codeword.

Example 6.4. *Suppose the input for the machine, figure 25, is the label* 010*. Begin at the start start and follow the arrows. First the machine inputs the rightmost bit* 0 *and outputs a* 0*, then the machine inputs the next bit* 1 *and outputs a* 0*, finally the machine inputs the rightmost bit* 0 *and outputs a* 1*. The final output is* 001*, which is the integer* 1 *represented in binary, and the machine ends at an accepting state. Therefore the label is a codeword corresponding to the integer* 001*, that is* 1*.*

Now suppose the input for the machine is the label 110*. Following the same procedure, the machine inputs a* 0 *and outputs a* 0*, then inputs a* 1 *and outputs a* 0*, and finally inputs a* 1 *and outputs a* 1*. The final output is* 001 *and the machine ends at state* 2*. Thus the label is not a codeword. Since the machine ends at state* 2*, the remainder is* 2 *and the integer corresponding to the nearest codeword is* 2*, that is* 1 *remainder* 2 *rounded up.*

There is a similar combination of machines which performs ENCODE on K_2^n.

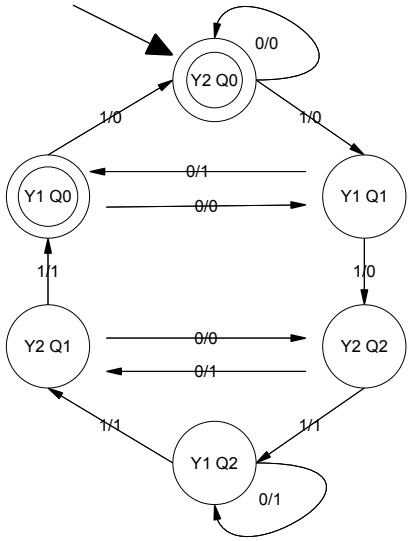

FIGURE 25. The FST which performs DECODE on K_2^n.

Unfortunately, this encoding and decoding scheme does not generalize to higher dimensions which are powers of two for the same reason that encoding and decoding for the Towers of Hanoi did not generalize. In the general case, some codewords are not adjacent to any multiple of $d+1$, while others are adjacent to two multiples of $d+1$.

It also seems that this encoding and decoding scheme will not generalize to other even dimensions as well, although further work would need to be done to check this.

6.4. **Hamiltonian Paths.** Another approach to take when looking for a simple encoding and decoding scheme on K_d^n is the use of Hamiltonian paths and circuits. We think Hamiltonian paths could be constructed on the iterated complete graphs to create a Gray labeling, so that every $d+1$ vertex corresponds to to a codeword.

Definition 6.5. *A **Hamiltonian path** is a path in a graph which visits each vertex exactly once. A **Hamiltonian circuit** is a cycle in a graph which visits each vertex exactly once and also returns to the starting vertex.*

Before using Hamiltonian paths and circuts on these iterated complete graphs, we must first prove that they do exist for all K_d^n.

Proposition 6.6. *For any pair of corner vertices v_1 and v_2 of K_d^n there is a Hamiltonian path from v_1 to v_2. Further, if $d > 2$, K_d^n has a Hamiltonian circuit.*

Proof. For $n = 1$, the claim is obvious because K_d^1 is a complete graph and thus has a Hamiltonian path between any pair of vertices, and except when $d = 2$ (the straight line), there is a Hamiltonian circuit. By its construction K_d^n is a complete graph whose vertices are d copies of K_d^{n-1}. Any vertex in K_d^n can be represented as (x,C) where C is one of the copies of K_d^{n-1} and x is a vertex within C. If (x,C) is a corner vertex of K_d^n, then x is also a corner vertex of C. Let $v_1 = (x_1,C_1)$ and $v_2 = (\hat{x}_d,C_d)$. Use the assumed Hamiltonian path of K_d^1 from C_1 to C_d to give an ordering $C_1,C_2,...,C_d$ on the copies of K_d^{n-1}. Complete the Hamiltonian path of C_1 from x_1 to the corner $\hat{x}_1$ which is adjacent to a corner of C_2 and call this corner x_2. Similarly, use the Hamiltonian path of C_2 from x_2 to $\hat{x}_2$ where $\hat{x}_2$ is adjacent to a corner of C_3. Continue this construction and finally use the Hamiltonian path of C_d to end up at $\hat{x}_d$. For the Hamiltonian circuit, assume $d > 2$, so that K_d^1 has a Hamiltonian circuit. Use this Hamiltonian circuit to put an ordering $C_1,C_2,...,C_d$ on the d copies of K_d^{n-1}. The pick pairs of corner vertices v_i and $\hat{v}_i$ for each C_i, so that v_i is adjacent to a corner vertex $\hat{v}_{i-1}$ of C_{i-1} and $\hat{v}_i$ is adjacent to a corner vertex of C_{i+1}. (Of course, C_{d+1} is C_1 and C_d is C_{i-1}.) Then use the above Hamiltonian path construction to connect the Hamiltonian paths of the C_i's into a Hamiltonian circuit of K_d^n. □

The easiest case for this approach is K_2^n. The graph of K_2^n is a straight line, so the Hamiltonian path is trivial. The Hamiltonian path creates a Gray labeling, in this case known as the reflected binary Gray labeling, where every third vertex corresponds to a codeword. This method corresponds to the same finite state machines created in section 6.3.

The next case to look at is K_3^n. There is indeed a Hamiltonian path on K_3^n which creates a Gray labeling where every fourth vertex corresponds to a codewod. This Hamiltonian path on K_3^3 is shown in figure 26. The construction of the path invovles a simple permutation. Looking only at the low order bit of each label, construct the path by following the permutation $0,1,2,2,1,0$.

The Recursive construction of this Hamiltonian path is very simple. It uses the fact that K_3^n is composed of d copies of K_3^{n-1}. First, begin with the Hamiltonian path that follows this permutation on the graph K_3^1. Then if n is even, on K_3^n construction the path clockwise through each K_3^{n-1}, beginning with the $0th$ copy, using the Hamiltonian path of K_3^{n-1} on each subgraph. If n is odd follow the same construction, but construct the path through each K_3^{n-1} counterclockwise, beginning with the $0th$ copy. Figure 27 shows this construction.

To encode and decode for K_3^n using the Gray code created by the Hamiltonian path:

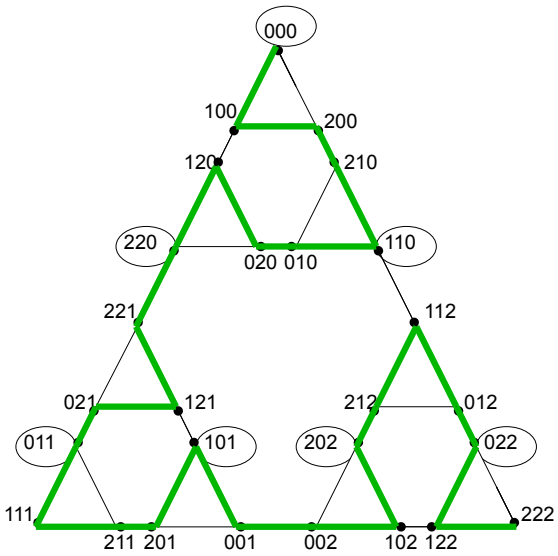

FIGURE 26. The thick line represents the Hamiltonian path on K_3^3. Notice that every forth vertex corresponds to a codeword.

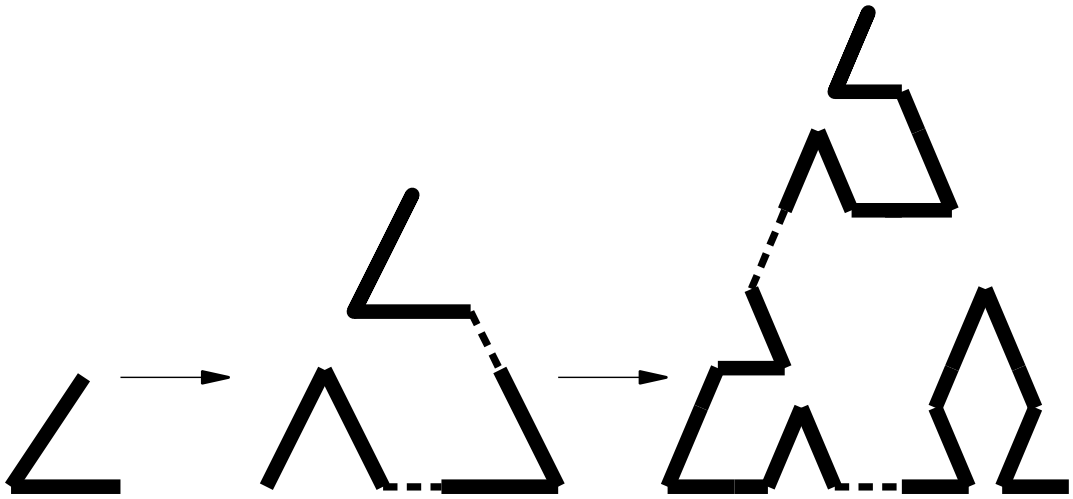

FIGURE 27. The construction of the Hamiltonian Path for K_3^1, K_3^2, and K_3^3

- CODE is given by:

$$CODE(I) = HAMILTONIANGRAY(4 * I)$$

- DECODE is given by:

$$DECODE(J) = TERNARY(J)/4$$

where J is a label of a codeword in the Hamiltonian Gray code, and TERNARY(J) converts this label from the Gray code, to its corresponding position in base 3

There exists a function that takes as input a vertex label and outputs the ternary number corresponding to the position of the vertex with this label in the Hamiltonian path. This function is done by the finite state machine in figure 28. From Theorem 6.3 there exists a finite state machine that divides numbers in base 3 by 4, which is shown in figure 21. Then it is possible to combine these two machines to create a finite state machine which performs DECODE on K_3^n, which is shown in figure 29.

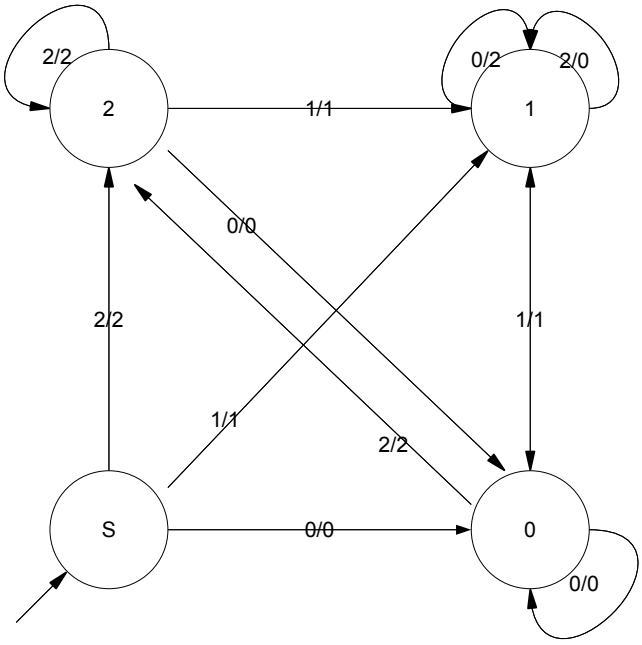

FIGURE 28. The FST that inputs a Gray code created by the Hamiltonian path on K_3^n and outputs the ternary number corresponding to the position of the Gray code in the Hamiltonian path. That is given J in Gray code starting with the high order bit (which corresponds to the rightmost bit of the Hamiltonian path created Gray code in the labeling of K_3^n), this machine computes TERNARY(J).

The finite state machine which performs DECODE on K_3^n works similarly to the finite state machine which performs DECODE for K_2^n. The input for

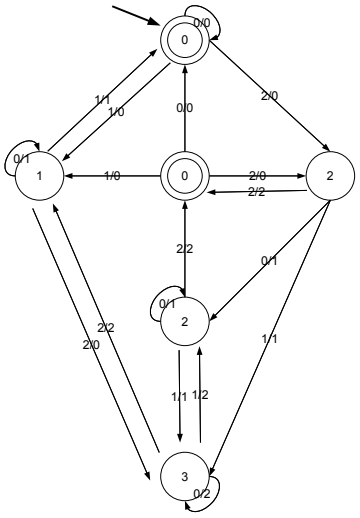

FIGURE 29. The FST which performs DECODE on K_3^n.

this machine are the codewords on K_3^n, that is G_n. The machine reads a string bit by bit, from right to left, following the transition with the given bit on the left side of the slash. The machine then changes that digit to the digit on the right side of the slash. The output of the machine is an integer in ternary which is an element of $\{0,1,...,|G_n|-1\}$, and corresponds to the correct quotient. There are two accepting states in the machine. If the machine ends at one of these accepting states, the label is a codeword, that is it corresponds to one of the labels which occurs every fourth vertex of the Hamiltonian path. If the machine ends at another state the label is a noncodeword, and that state corresponds to the remainder.

Example 6.7. *Suppose the input for the machine, figure 29, is the label* 220*. Begin at the start state and follow the arrows. First the machine takes the rightmost digit* 0 *and outputs a* 0*, then the machine inputs the next digit* 2 *and outputs a* 0*, finally the machine inputs the leftmost digit* 2 *and outputs a* 2*. The final output is* 002*, which is the integer* 2 *represented in ternary, and the machine ends at an accepting state. Therefore, the label is a codeword corresponding to the integer* 2*.*

Now suppose the input for the machine is the label 112*. Following the same procedure, the machine takes a* 2 *and outputs a* 0*, then takes a* 1 *and outputs a* 1*, and finally takes a* 1 *and outputs a* 2*. The final output is* 012 *and the machine ends at state* 3*, which is not an accepting state. Thus the label is not a codeword.*

Recall that the purpose of using Hamiltonian paths was to create a simple way to encode and decode. We have been able to find a simple path that does this for dimension 3 puzzles. Because we want to generalize this method we tried to find a Hamiltonian path in the dimension 5 puzzles. Because the permutation 012 and then its reverse, 210, had worked for dimension 3 puzzles we hoped this would generalize to dimension 5. Unfortunately, this did not work. We also attempted to follow various permutations within the subgraphs but this did not work due to the parities before entering the subgraph. Further investigation should be done on other possible paths that would not follow a permutation pattern.

7. Distance Problem

Encoding and decoding methods, error-correcting machines, error recognizing machines and Gray code property are all attributes that make a labeling more attractive. Notice that the SF labeling and the Dimension 2^m labeling have some of these attributes, which are mentioned in previous sections. Another attribute that would add more attraction to these labelings would be a finite-state transducer that would calculate the distance from any labeling to another. Previous work done by Kind [5] tells us there is an algorithm which calculates this distance for the SF labeling. We wanted to find a finite-state transducer that carries out a similar function.

Definition 7.1. *A **box** is a K_d^{n-1} within K_d^n. For any string x and y the shortest distance is given by **d(x,y)**.*

It should be clear that $d(x,y) \leq 2^n - 1$ in G_d^n and the distance between any two corner vertices of G_d^{n-1} is $2^{n-1} - 1$. Consider x from box A and y from box B. Then we have three possibilities to consider:

(1) Going from x to a corner of A which is adjacent to box B. Then going to box B and then to y.
(2) Going from x to some corner of A which is adjacent to another box, call it box C. Then going to a corner of box C that is adjacent to box B. Then going to box B and eventually to y.
(3) Going from x to some corner of A. Then passing through 2 or more boxes to get to box B and eventually to y.

The next step is to observe which cases will give us the shortest distance between x and y. We assign subscripts to represent corners of a box. Each case has a unique distance formula which is given below.

Case 1: Passing through 0 boxes:

$$d(x,A_1) + 1 + d(B_1,y) < 2^n$$

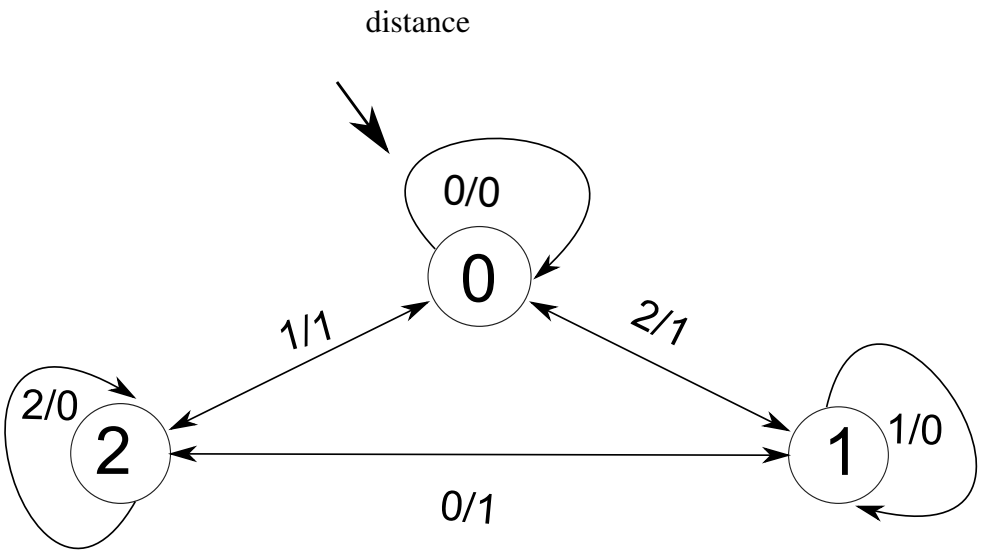

FIGURE 30. Finite-state transducer that outputs, in binary, $d(0...00,x)$ for any configuration x in the standard Towers of Hanoi puzzle.

Case 2: Passing through 1 box:

$$d(x,A_2)+1+d(C_1,C_2)+1+d(B_2,y)=d(x,A_2)+2^{n-1}+1+d(B_2,y)$$

Case 3: Passing through 2 boxes:

$$d(x,A_3)+1+d(C_1,C_2)+1+d(D_1,D_2)+1+d(B_3,y)=d(x,A_2)+2^n+1+d(B_2,y)$$

From the above calculations we can see that passing through 2 or more boxes will give a longer path than passing through 0 or 1 boxes. But passing through 0 boxes may not be shorter than passing through 1 box. For example, let x = 201 and y = 110 from the Towers of Hanoi K_3^3. Then the path that passes through 0 boxes has length 7 and the path passing through 1 boxes has length 6. Therefore, passing through 0 boxes may not be the shortest path. Note, for the spin-out puzzle calculating the distance is very simple because there is only one path. By converting the configurations into binary and subtracting them will give us the distance. To compare Case 1 and Case 2 we need a way to calculate the distance of any vertex in a box to a corner of that box. This will proved the extra information that is necessary to find the shortest distance. We observe for only the standard Towers of Hanoi puzzle. Figure 30 shows the finite-state transducer that calculates the distance, in binary, to the corner 0...00. NOTE: the strings will be read right to left when inputting.

Due to symmetry a few permutations will give us the machine that will calculate the distance to 1...11 or to 2...22. All we need to do is change the starting state. The machine that calculates $d(1...11,x)$ is the same machine in figure 30 but the starting state is at 1 and to calculate $d(2...22,x)$ we change the starting state to 2.

Example 7.2. *In K_3^3 let x = 012. Since we read right to left we input 210, starting at state 0 the output would be 101 in binary which is a 5 in decimal. If we start at state 1 the output would be 111 which is 7 in decimal. And if we start at state 2 the output would be 010 which is 2 in decimal. Observe, they all give the correct output as desired.*

Our hope is that similar machines exist for all dimensions. Since a simple machine exists for dimension 3 the next step would be to observe for dimension 5, with the SF labeling.

8. Conclusions

Using previous findings we were able to construct a recursive, iterative and count algorithm for the SF puzzles and a recursive and iterative algorithm for the dimension 2^m puzzles. In addition, using the rules of the combination puzzle we were able to construct the dimension 10 and dimension 12 labeling as shown in Section 5. We next attempted to find a simple method of encoding and decoding. We have shown the encoding and decoding scheme for the Spin-out puzzle. Unfortunately, this scheming method did not extend to the general dimensions. In search for a simple scheme we began observing Hamiltonian paths. As shown in Section 6 there does exist a construction of a Hamiltonian path for the Towers of Hanoi puzzle that does the encoding and decoding, as desired. But a similar construction would not work for dimension 5. Lastly, we wanted to find a construction of a finite-state machine that would calculate the distance of any two configurations. This was easily done for the Towers of Hanoi puzzle, as shown in Section 7, and would likely extend to further dimensions.

Further research on these iterated complete graphs will include:

(1) Finding a count algorithm for the dimension 2^m puzzles.
(2) Find a construction for all even dimensions.
(3) Find a simple encoding and decoding scheme for the SF labeling and dimension 2^m labeling. We believe using Hamiltonian paths will be helpful.
(4) Find a general construction of a finite-state machine to calculate distance between two configurations.

REFERENCES

[1] Elizabeth Stubak and Nickholas Stevenson. *A New Puzzle for Iterated Complete Graphs of Any Dimension*. 2008.

[2] David Bode. *Alternate Labelings for Graphs Representing Perfect-One-Error-Correcting Codes*. 1998.

[3] Elizabeth Weaver. *Gray Codes and Puzzles on Iterated Complete Graphs*. 2005.

[4] Stephanie Kleven. *Perfect Codes on Odd Dimension Serpinski Graphs*. 2003.

[5] Kathleen King. *A New Puzzle Based on the SF Labelling of Iterated Complete Graphs*. 2004.

[6] Ingrid Nelson. *Coding Theory on the Towers of Hanoi*. 1995.

[7] Paul Cull and Ingrid Nelson. *Perfect Codes, NP-Completeness, and Towers of Hanoi Graphs*. Bull. Inst. Combin. Appl. 26: 13–38, 1999.

[8] Paul Cull and Ingrid Nelson. *Error-correcting codes on the towers of Hanoi graphs*. Discrete Math., 208/209: 157-175, 1999.

[9] Harry R. Lewis and Christos H. Papadimitriou *Elements of the Theory of Computation, Second Ed.* Prentice-Hall, New Jersey, 1998.

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

[11] Kirk Pruhs. *The SPIN-OUT Puzzle*. ACM SIGCSE Bulletin, 25(3):36–38, 1993.

[12] Erwin Engeler. *Introduction to the Theory of Computation*. Academic Press, New York, 1973.

[13] Pamela Russell. *Perfect One-Error-Correcting Codes on Iterated Complete Graph*. 2004.

[14] Robin J. Wilson. *Introduction to Graph Theory*. Longman, Malaysia, 1996.

Acknowledgments: Much of the work here was carried out by students in the REU Summer Program at Oregon State University in previous years. We would like to thank the following people for their contributions: Elizabeth Skubak, Nicholas Stevenson, Ingrid Nelson, Jessica Cavanaugh, Kevin Stoller, David Bode, Be Birchall, Jason Tedor, Shaun Alspaugh, Nathan Knight, Kathleen Meloney, Christopher Frayer, Shalini Reddy, Stephanie Kleven, Kathleen King, Pamela Russell, and Elizabeth Weaver. Many of their papers appear on the website:
`http://math.oregonstate.edu/~math_reu/REU2008/`.

COLLEGE OF ST. BENEDICT
E-mail address: `labaun@csbsju.edu`

CALIFORNIA STATE POLYTECHNIC UNIVERSITY
E-mail address: `chauhan@csupomona.edu`