

A NEW PUZZLE FOR ITERATED COMPLETE GRAPHS OF ANY DIMENSION

ELIZABETH SKUBAK AND NICHOLAS STEVENSON

ADVISOR: PAUL CULL
OREGON STATE UNIVERSITY

ABSTRACT. The Towers of Hanoi puzzle can be used to label a family of graphs in a way that provides easy (finite-state) coding properties. This puzzle can be modified to have any odd number of towers (greater than three) and the rules adjusted so that similar easy labelings can be created on the corresponding odd-dimensional graphs. However, the puzzle cannot be directly extended to an even number of towers and retain its essential structure. Are there other puzzles that exist on even-dimensional graphs in this family? A puzzle for the 2-dimensional case is known, and is even sold commercially as the Spin-Out Puzzle. We give an extension of this puzzle for graphs of any dimension 2^m for $m \geq 1$. We also explore combinations of this extended Spin-Out Puzzle with the Towers of Hanoi puzzles to create puzzles that correspond to the Sierpinski graphs for any dimension. In addition, we present two new labelings which have simple constructions and some easy coding properties.

1. INTRODUCTION

The Towers of Hanoi puzzle is an interesting and often-studied puzzle that also has curious applications to coding theory. In particular, it can be used to label a family of graphs in a way that provides easy (finite-state) coding properties[10]. In Section 2, we give background information on this family, called Sierpinski graphs or iterated complete graphs, as well as on codes and labels. It has also been proven in previous work (see [16],[14]) that the Towers of Hanoi puzzle can be modified to have any odd number of towers (greater than three) and the rules adjusted so that similar easy labelings can be created on the corresponding odd-dimensional graphs. We summarize these results in Section 3.

However, the puzzle cannot be directly extended to an even number of towers and retain its essential structure. Are there other puzzles that exist on these associated even-dimensional graphs? A puzzle for the 2-dimensional case is known, and is even sold commercially as the Spin-Out Puzzle, which we explore in Section 4. In Section 5, we give an extension of this puzzle to the 4-dimensional case based on work by Weaver[29], and also extend this to a puzzle for graphs of any dimension 2^m for $m \geq 1$ in Section 6. We then give the recursive labels of the graphs corresponding to these puzzles, and show that there are finite-state machines that can recognize codewords as well as error-correct words. In Section 7, we explore combinations of the extended Spin-Out Puzzle with the Towers of Hanoi puzzles to create new puzzles that correspond to the iterated complete graphs for all dimensions, including the remaining even dimensions. These are created by writing any dimension as $q \cdot 2^m$ for q odd; the puzzle will have q towers but also pieces

Key words and phrases. graphs, codes, puzzles, Sierpinski graphs, iterated complete graphs, error-correction.
This work was done during the Summer 2008 REU program in Mathematics at Oregon State University.

that have 2^m orientations on each tower. The rules of the puzzle are essentially a combination of the rules of both puzzles. As desired, they reduce to the rules for the known puzzles when the dimension is odd or a power of two.

Our search for other labelings produced two, the Corner-Distance and Subgraph labelings, which have intuitive constructions and several desirable characteristics. We stopped short of considering puzzles for them. Section 8 covers the labelings and some basic results.

CONTENTS

1. Introduction	1
2. Graphs, Labels, and Codes	3
2.1. Iterated Complete Graphs	3
2.2. Codes on Graphs: Perfect One-Error-Correcting Codes	4
2.3. Labelings and Codewords	4
2.4. The G-U Construction	4
3. The SF Labeling and Puzzle	6
3.1. Construction of the SF Labeling	7
3.2. The SF Puzzle	7
4. Dimension 2: The Spin-Out Puzzle	9
4.1. Graph and Label	10
4.2. Codeword Recognition	11
4.3. Error Correction	11
5. Dimension 4: The Reflection Puzzle	12
5.1. Rules	12
5.2. Graph and Label	13
6. Dimension 2^m	14
6.1. Rules	15
6.2. Graph and Labeling	16
6.3. Recursive Labeling	18
6.4. Codeword Recognition	20
6.5. Error Correction	23
7. Other Even Dimensions	26
7.1. Dimension 6	26
7.2. General Dimensions	28
7.3. Conclusions on the Puzzle for General Dimension	31
8. New Labelings	31
8.1. The Corner-Distance Labeling	32
8.2. The Subgraph Labeling	36
8.3. Conclusions on the New Labelings	38
References	38

2. GRAPHS, LABELS, AND CODES

In this section we give some background on the specific graphs we will be working with, as well as the basic definitions for codes on graphs. Together this information forms the basis for our investigations.

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 some complete graphs.

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

Definition 2.4. An **iterated complete graph**, also known as a **Sierpinski graph**[15], 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_d^n .

Iterated complete graphs are easier to explain using examples. 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_6^1 , K_6^2 and K_6^3 , illustrating how each graph is constructed from the graph of the previous dimension.

FIGURE 2. The iterated complete graphs K_6^1 , K_6^2 and K_6^3 .

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: Perfect One-Error-Correcting Codes.

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 **perfect one-error-correcting code** (or **PIECC**) on a graph G is a code such that:

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

Examples of PIECC's can be found in the left-hand graphs in Figures 3, 4, and 5.

2.3. Labelings and Codewords.

Definition 2.9. 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 2.10. 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 2.11. 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[10] proved that determining whether a given graph has a PIECC is an NP-complete (difficult) problem. However, they introduced a relatively simple method for constructing a PIECC 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 later found to generalize to higher dimensions, and independently by Klavzar, Milutinovic, and Petr in [15]. Cull and Nelson's method has come to be known as the G-U construction. It is the foundation of our methods for codeword recognition and error correction on iterated complete graphs.

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 .

This is much easier to understand via example. 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

Because labels and puzzles for odd dimensions have been established in previous papers (see [14], [16]), the focus of our paper is iterated complete graphs with even dimension. In this section

we summarize those results. The SF labeling on the odd dimension 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[10]. 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 $1, 2, \dots, (d-1)$ 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 counterclockwise, 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_7^2 . Figure 8 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 9 shows the SF labeled graph K_3^3 corresponding to Towers of Hanoi with 3 disks.

Now imagine we have an odd number $d \geq 3$ of towers numbered 0 through $d-1$. Like the Towers of Hanoi, configurations of n disks on these towers can be represented by base d strings of length n . The SF puzzle has the same rules as Towers of Hanoi. In addition, it has the following rules to restrict the possible moves to those represented by the SF labeling on K_d^n .

- (1) No disk may be moved unless all of the disks smaller than it are stacked together on the same tower.

FIGURE 7. The SF labeling of K_7^2 .

- (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 $(2a - b) \bmod d$.

Note that the movement of the smallest disk is unaffected by these additional rules; it can always move to any tower. Figure 10 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 indeed there is an edge between these two vertices in K_5^3 , as shown in Figure 8.

FIGURE 8. The SF labeling of K_5^3 .

FIGURE 9. The labeled graph K_3^3 corresponding to the Towers of Hanoi with 3 disks.

4. DIMENSION 2: THE SPIN-OUT PUZZLE

A good starting point for our investigation into puzzles for even-dimensional iterated complete graphs is the Spin-Out puzzle by ThinkFun[26]. 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

FIGURE 10. Configurations corresponding to labels 220 and 224 on K_5^3 . The largest disk may move between towers 0 and 4.

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 11. A configuration of the Spin Out puzzle. 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's 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 , which we now describe.

4.1. Graph and Label. The reflected binary Gray code is a well-known labeling scheme on K_2^n with an easily defined recursive construction[27].

For $n = 1$, let the reflected binary Gray code, G_1 , be 0-1. That is, we label one vertex 0 and the other vertex 1.

To construct G_n :

- (1) Take two copies of G_{n-1} . Append 0 to the labels in the first copy and 1 to the labels in the second copy.

FIGURE 12. The reflected binary Gray code for $n = 1$, $n = 2$, and $n = 3$ with code vertices circled

- (2) Reverse the order of the strings in the second copy, and connect the new beginning of the second copy to the end of the first copy.

Note that in some papers, these 0's and 1's are sometimes prepended rather than appended; we append for consistency with previous work and for their relationships to the puzzles.

4.2. Codeword Recognition. If we consider the perfect one error-correcting code on this labeling as defined by the G-U construction (see Section 2.4), we find the code vertices as shown circled on the graphs. Now, given a label, we would like to know whether or not it is a codeword.

More specifically, is there a finite-state machine that recognizes codewords? Yes, there is a machine that can be created directly from the recursive construction of the label and code. This codeword recognizer for the reflected binary Gray code is shown in Figure 13. This machine is nondeterministic, since it has more than one start state. If the iteration is even, the machine starts in state G^e . If the iteration is odd, the machine starts in state G^o . If the machine ends in the accepting state, then the word is a codeword.

We can easily convert this machine to a deterministic machine, though we do not show this here (see [21]) for theory on finite-state machines). Later, (Section 6.4, we will prove a more general case of this recognizer and show a deterministic version. Alternatively, if we require that $0 \dots 0$ is a codeword, the only PIECC on the graphs K_2^n has the codewords at every third vertex. Thus this machine only needs to convert the Gray label to its binary position and determine if that position is a multiple of 3, both of which can be simply done with finite-state machines.

4.3. Error Correction. If we place a word into the recognizer, we can decide whether or not it is a codeword. One motivation for perfect one-error-correcting codes, however, is to be able to easily find, for any word, the closest codeword. This is called error correcting.

The error corrector for the reflected binary Gray code labeling is shown in Figure 14. The machine reads a string bit by bit, following the transition with the given bit on the left part of the label. The machine then changes that bit to the bit on the right side of the label. The codeword recognizer above must first be used. If the final state is a G state, the error corrector begins at the 0 state (which returns the string unchanged, since it is a codeword). If the final state is either of the C_1 states, the machine begins in 1; if the final state is a C_2 state, it begins in 2.

The proof that a generalized version of this error corrector indeed works can be found in Section 6.4.

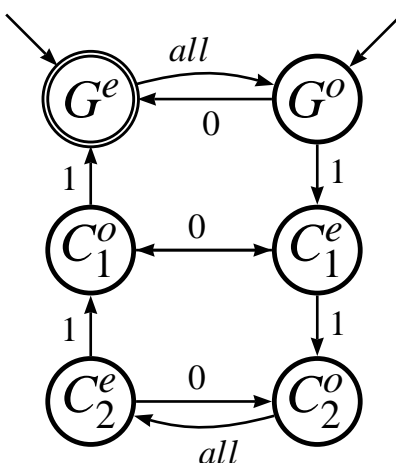


FIGURE 13. Finite-state Recognizer for the Spin-Out Puzzle

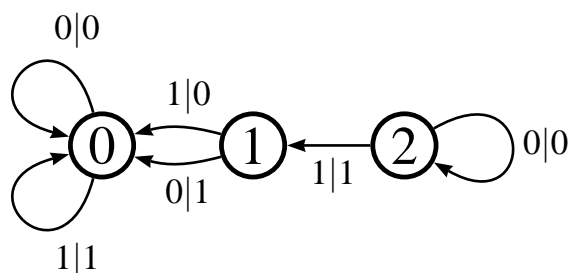


FIGURE 14. Error-corrector for the Spin-Out Puzzle

5. DIMENSION 4: THE REFLECTION PUZZLE

The Reflection Puzzle was first described by Weaver[29]. We found a physical representation of this labeling of the iterated complete graphs of dimension 4. It bears a close resemblance to the Spin-Out puzzle. In fact, it is essentially two such puzzles stacked on top of each other. Our goal will still be to change each piece to the zero orientation. In this puzzle, however, each *piece* is made up of two *spinners* that can (sometimes) move independently, giving four possible orientations, shown in Figure 15.

FIGURE 15. Spinner orientations for the Reflection Puzzle

We will again use the labels of the pieces to represent puzzle configurations. An example can be found in Figure 16.

5.1. Rules. As usual, the leftmost piece may move at any time. Here, however, it may also move to any orientation, allowing 3 possible moves. For a piece $j \neq 0$ to be able to move, we must have

FIGURE 16. An example puzzle orientation for the Reflection Puzzle labeled 0320

pieces 0 through $j - 2$ at orientation 0. Also, if piece $j - 1$ is at 0, then no spinners in piece j can move; thus we need $f(j - 1) \neq 0$.

So suppose piece j is able to move. Which orientation should it change to? Our rule is: if a spinner can move, it does move. Thus for any vertical spinner in piece $j - 1$, the corresponding spinner in j will change orientations. In Figure 16, the second to last piece may change from 2 to 1.

5.2. Graph and Label. To represent this puzzle on a graph, we will associate the labels as defined above with vertices and legal moves with edges. As hoped, the graph generated is K_4^n . Since only one piece is moved at a time in the puzzle, this new labeling is a Gray labeling.

To create this labeling when $n = 1$, the top vertex of K_4^1 is labeled 0 and each other vertex, moving counterclockwise, is labeled with the successive integers through 3.

To construct this labeling scheme on K_4^n for $n > 1$:

- (1) Construct 4 copies of the labeling on K_4^{n-1} , and index these copies from 0 to 3.
- (2) Reflect copy 1 vertically, and add a 1 to the end of every label in the copy.
- (3) Reflect copy 2 both vertically and horizontally, and add a 2 to the end of every label in the copy.
- (4) Reflect copy 3 horizontally, and add a 3 to the end of every label in the copy.
- (5) Place copy 0 as the top copy and the others in order counterclockwise. Finally connect each copy to all of the other copies, as shown in Figure 17.

As in the Spin-Out puzzle and associated labeling, this labeling has finite-state recognition and error correction, which we prove generally in the next section.

FIGURE 17. The Reflection-Method labelings of K_4^1 and K_4^2 6. DIMENSION 2^m

The puzzle on four dimensions suggests 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. Note that all proofs in this section apply to the Spin-Out and Reflection puzzles. At present, these additional puzzles are mathematical constructions and they may be difficult to physically construct so that all the mathematical rules are completely enforced.

We need a way to associate orientations of the pieces with numbers 0 through $d - 1$. We define the orientations of our 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 orientation 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 $j \in \{0, \dots, d - 1\}$ we have a distinct orientation and corresponding binary number. Since there are exactly d orientations, we know we have defined all possibilities.

Example 6.1. 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 18. Piece orientations for the Dimension 8 Puzzle

Note that the 0^{th} orientation will always consist of all horizontal spinners. Note also that Spin-Out and the Reflection Puzzle both consistently follow this naming scheme.

Now, for an iteration n for $n \geq 1$, we will have n puzzle pieces. We will 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 our puzzle, we will label it with a string of characters from $\{0, \dots, d-1\}$, where each piece 0 through $n-1$ is represented by the number of the orientation it is in. To avoid confusion, we will write $f(j)$ to refer to the orientation of piece j .

Example 6.2. *Continuing from the example above, Figure 19 has the label 0374.*

FIGURE 19. An example configuration for the Dimension 8 Puzzle. The pieces are numbered left to right 0, 1, 2, and 3.

6.1. Rules. The rules of this puzzle are nearly the same as for the Reflection Puzzle. First, the 0^{th} piece may always change orientation, and may change to any other orientation.

To spin at least one spinner of the j^{th} piece, we must have that pieces 0 through $j-2$ are 0 and piece $j-1$ is not 0 . In our notation, we need $f(0)$ through $f(j-2)$ to be 0 and $f(j-1) \neq 0$. If these conditions are satisfied, then we must 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.

For example, in Figure 19, 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 we may change the orientation of piece 2 from 7 to 4.

The goal of the puzzle is, given some initial configuration, to move all the pieces to orientation 0 . Since this puzzle can be represented by K_d^n as we prove shortly, we know that the minimum solution path for any two configurations is not longer than $2^m - 1$, the diameter of the graph. This quantity is the same as for both the Spin-Out puzzle and the SF Puzzles. Another way of “solving” this puzzle might be to choose two configurations and move from one to the other. Of course,

puzzles are fun, but the focus of the paper is the puzzle's relationship to the family of iterated complete graphs, which we will now prove, after some preliminary lemmas.

It would be helpful to be able to know which orientation piece j can move to without thinking in terms of spinners. To do this, we define a special operation. 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).

Lemma 6.3 (Orientation Change Function). *If $f(0)$ through $f(j-2)$ are 0 and $f(j-1) \neq 0$, then piece j may move to $f(j-1) \oplus f(j)$. Because $f(j-1) \oplus f(j) \neq f(j)$, piece j can make a change to another distinct orientation.*

Proof. Recall that we defined our orientations to coincide with binary numbers representing the spinners in a piece; a 0 is a horizontal spinner, and a 1 is a vertical spinner. Then consider that movement of the spinners of piece j are completely dictated by the positions of the spinners of piece $j-1$. In particular, any horizontal spinners in piece $j-1$ block movement in piece j of the spinner in the same horizontal plane. Therefore, any 0's in the binary representation of $f(j-1)$ dictate that no change should occur in the corresponding bits in the binary representation of $f(j)$. Since $0 + x = x$ in bitwise addition, our formula holds in this case.

For the remaining case, consider the vertical spinners in $j-1$, i.e. the 1's in the binary representation of $f(j-1)$. These allow movement of the corresponding spinners in j and according to our rule, the spinners must change. Since $1 + 1 = 0$ and $1 + 0 = 1$ in bitwise addition, our formula holds for each bit. Thus the result $f(j-1) \oplus f(j)$ is the orientation to which piece j may move.

Finally, note that since $f(j-1) \neq 0$, it has at least one bit equal to 1, which causes a change in that bit in $f(j)$, so $f(j-1) \oplus f(j) \neq f(j)$. \square

Note that $f(j-1) \oplus f(j)$ can equal $f(j-1)$. In fact, this occurs if and only if $f(j) = 0$. Thus the configuration $0 \dots 0x0$ may move to $0 \dots 0xx$.

Lemma 6.4 (Reversibility of Moves). *All moves in this puzzle are reversible.*

Proof. Clearly all moves by the leftmost piece are reversible since we may move this piece at any time. To consider moves by some other piece, first let $x \in \{1, \dots, d-1\}$ and $y \in \{0, \dots, d-1\}$. Also, let W be the empty string or some fixed string consisting of characters from $\{0, \dots, d-1\}$. Now suppose the configuration $0 \dots 0xyW$ may move to $0 \dots 0xzW$ (where there may be any fixed nonnegative number of leading zeros).

By Lemma 6.3, $z = x \oplus y$. Then in $0 \dots 0xzW$, the piece at z may change to $x \oplus z = x \oplus (x \oplus y)$. It is easy to check that bitwise addition (the XOR) is associative. Also, note that any string added bitwise to itself is simply the zero string since $1 + 1 = 0 + 0 = 0$. Thus $x \oplus (x \oplus y) = (x \oplus x) \oplus y = 0 \oplus y$, which simply equals y . Therefore the configuration $0 \dots 0xzW$ may move to $0 \dots 0xyW$ as desired. \square

Thus we know that \oplus is associative as well as commutative, and that each number is its own inverse under \oplus . These properties will be of further use to us.

6.2. Graph and Labeling. Now that we have a puzzle, we wish to show that it is in fact the puzzle we were looking for—the puzzle that fits the graph K_d^n (for $d = 2^m$). We label each vertex in K_d^n with the label of some puzzle configuration and connect two vertices if the two configurations are one legal move apart. Since we can only move one piece at a time, the labeling scheme is a

Gray code. Also, by Lemma 6.4, the edges are not directed; we may always undo the last move we made. The following lemmas will help in our proof.

Lemma 6.5 (Corner Labels). *The puzzle configurations $0\dots 0k$ for $k = \{0, \dots, d-1\}$ have exactly $d-1$ adjacent configurations (where there may be any nonnegative number of leading zeros).*

Proof. By a rule of our puzzle, the 0^{th} piece, at orientation 0, may always change to any other orientation $1, \dots, d-1$. This gives us $d-1$ possible moves. To see that there are no others, note that there is no piece where $f(j-1) \neq 0$. Since this is required for a piece $j \neq 0$ to move, no other moves are possible. \square

Lemma 6.6 (Non-Corner Labels). *All other configurations (those not of the form $0\dots 0k$ for $k = \{0, \dots, d-1\}$) have exactly d moves.*

Proof. First, note that any configuration not of the form $0\dots 0k$ for $k = \{0, \dots, d-1\}$ instead has the form $0\dots 0xyW$ where there may be any nonnegative number of leading zeros, $x \neq 0$, and where W is the empty string or some string consisting of characters from $\{0, \dots, d-1\}$. Again, the first piece may always change to any other orientation $1, \dots, d-1$, giving us $d-1$ possible moves. Now, we show that there is exactly one more. Since $x \neq 0$, the piece j such that $f(j) = y$ is the only piece that has $f(0)$ through $f(j-2)$ equal to 0 and $f(j-1) \neq 0$. Therefore piece j is the only other piece that may move. By Lemma 6.3, piece j may change from orientation y to some other distinct orientation, giving us a total of d moves. \square

Lemma 6.7. *For a fixed $y \in \{0, \dots, d-1\}$, the function defining the possible change of the character y in the configuration $0\dots 0xy\dots$ of domain $x \in \{1, \dots, d-1\}$ is a bijection onto $\{0, \dots, d-1\} \setminus \{y\}$. That is, for any other character z not y , there is some nonzero x such that $0\dots 0xy\dots$ may change to $0\dots 0xz\dots$.*

Proof. First, let the piece with orientation y be piece j . Since $x \neq 0$, y does move, and so this function is well-defined. By Lemma 6.3, piece j actually changes, so y is not a valid element of the range; thus the domain and range are the same size. Now we need only show that this function is injective. So suppose that both x_1 and x_2 change y to z . That is, $x_1 \oplus y = z = x_2 \oplus y$. Bitwise adding y shows that $x_1 = x_2$. \square

Theorem 6.8. *The described generalized Spin-Out puzzle fits the graph K_d^n . That is, each vertex in K_d^n represents a configuration of pieces, and each edge represents a valid move between two configurations.*

Proof. We will use induction on the number of puzzle pieces n , which will correspond to the iteration of the graph K_d^n . First consider the puzzle with one puzzle piece consisting of m spinners. Because this piece is the 0^{th} piece, it may change to any orientation at any time, and because it is the only piece, these are the only moves we can make. Clearly this corresponds to the complete graph $K_d = K_d^1$.

Now assume that the puzzle with $(n-1)$ pieces corresponds to the graph K_d^{n-1} . We will consider the puzzle with n pieces. Note that if we may move the j^{th} piece in some puzzle, the $(j+1)^{\text{st}}$ piece (and any pieces farther right) is essentially irrelevant. As seen in the example in Figure 19, the last piece does not affect what moves are legal by piece 0 through piece 2. Thus we consider the puzzle with n pieces to be a combination of d puzzles, each with $n-1$ pieces, one puzzle for each

possible orientation of our added piece. Therefore, one character representing the orientation of the new piece will be appended to each label. The d smaller puzzles are each copies of K_d^{n-1} by our induction hypothesis. Then all that remains is to show that these d subgraphs connect in the correct manner.

Recall that an iterated complete graph of dimension d has d corner vertices (vertices that are degree $d - 1$). We claim that exactly one vertex from each of the K_d^{n-1} subgraphs will become a corner vertex of K_d^n : the vertex $0 \dots 0$, which we know is a corner vertex by Lemma 6.5. To see this, note that in K_d^n , the label $0 \dots 0$ will be appended with some $r \in \{0, \dots, d - 1\}$. Simply using Lemma 6.5 once more shows that the vertex $0 \dots 0r$ is a corner vertex of K_d^n , giving us d such corner vertices. Clearly these are the only ones, since any other vertex of K_d^{n-1} , when appended by some $r \in \{0, \dots, d - 1\}$, cannot have the form $0 \dots 0r$. By Lemma 6.6, these other vertices are not a corner vertices, and we have verified this claim.

Next we show that every other corner of each K_d^{n-1} subgraph connects to exactly one different K_d^{n-1} subgraph at one of its corner vertices. Each of these other corner vertices in the K_d^{n-1} graphs have the form $0 \dots 0s$ for $s \in \{1, \dots, d - 1\}$ (Lemma 6.5). Thus, in K_d^n , their new labels have the form $0 \dots 0st$ for $s \in \{1, \dots, d - 1\}$ and $t \in \{0, \dots, d - 1\}$. Then $0 \dots 0st$ may move to $0 \dots 0s(s \oplus t)$, which is in the $(s \oplus t)^{\text{th}}$ subgraph; by Lemma 6.3 this is a different subgraph than the t^{th} subgraph.

So consider some fixed K_d^{n-1} subgraph, i.e. fix t in $0 \dots 0st$. By Lemma 6.7, over all values of s (we know $s \neq 0$), t will change to every character except t itself. This implies that this fixed subgraph must be connected to each of the other $d - 1$ subgraphs exactly once. Since t was arbitrary, this is true for each K_d^{n-1} subgraph. This finishes our proof. \square

6.3. Recursive Labeling. Theorem 6.8 tells us how to construct our graph and labeling from the puzzle. We wish, however, to have a way to recursively construct the labelings on the family of iterated complete graphs K_d^n without referring to the puzzle.

Unfortunately, this labeling of the family of iterated complete graphs of these dimensions is not unique. For at least dimension 8, there exist other labelings that still preserve the desired property of having the reflected binary Gray code along each diagonal from $0 \dots 0$. However, it does not appear that any puzzle easily associates with any other labeling we have found.

The base case for our labeling is the complete graph $K_d = K_d^1$. Clearly any labeling of the vertices would correspond to the puzzle, so without loss of generality we will label some ‘‘top’’ vertex 0 and label in order counterclockwise, and call this labeling L_d^1 .

Now assume we have L_d^{n-1} . We know that L_d^n is based on d copies of L_d^{n-1} by Theorem 6.8; however, if we simply placed each copy down, the edges connecting these subgraphs would not draw the graph as we have neatly depicted it in this paper. Therefore we will permute each subgraph so that the edges are in the desired locations. Clearly without loss of generality, we may place the 0^{th} subgraph any way we wish, so we will not permute it. Then 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, $\Gamma_i(\dots x) = \dots (x \oplus i)$. 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.

Example 6.9. Figure 20 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 1st 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.

FIGURE 20. 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.

Lemma 6.10. Each permutation Γ_i , for any $i \in \{0, \dots, d-1\}$, is a graph automorphism on L_d^n . That is, applying any Γ_i does not change the structure or any edges of the graph.

Proof. First of all, note that Γ_i is in fact a permutation on the vertices. Indeed, suppose $\Gamma_i(\dots y) = \Gamma_i(\dots z)$. (Since Γ_i only changes the last character, we know that the rest of the label must be identical.) But $\Gamma_i(\dots y) = \dots(y \oplus i)$ and $\Gamma_i(\dots z) = \dots(z \oplus i)$. Then $y \oplus i = z \oplus i$ and $y = z$, so the labels $\dots y$ and $\dots z$ are the same. Thus Γ_i is injective, and since Γ_i maps a set to itself, Γ_i is a bijection.

Now, we show that for adjacent vertices u, v in L_d^n , we have that the vertices $\Gamma_i(u)$ and $\Gamma_i(v)$ are adjacent in $\Gamma(L_d^n)$. There are three cases. (In all cases, the number of leading zeros may be positive or zero.)

- (1) If the vertex is labeled $0\dots 0x$, then it is adjacent only to the $d-1$ vertices $y0\dots 0x$ for $y \in \{0, \dots, d-1\}$ (see Lemma 6.5). Call this set of vertices S . Now $\Gamma_i(0\dots 0x) = 0\dots 0(x \oplus i)$, which, again by Lemma 6.5, is adjacent only to $y0\dots 0(x \oplus i)$ for $y \in \{0, \dots, d-1\}$. The set of these vertices is clearly equal to $\Gamma_i(S)$.

(2) If the vertex is labeled $0 \dots 0xz$, then it is adjacent to only the vertices in the set

$$S = \{y0 \dots 0xz : y \in \{0, \dots, d-1\}\} \cup \{0 \dots 0x(x \oplus z)\}.$$

But $\Gamma_i(0 \dots 0xz) = 0 \dots 0x(z \oplus i)$, which is adjacent only to

$$\{y0 \dots 0x(z \oplus i) : y \in \{0, \dots, d-1\}\} \cup \{0 \dots 0x(x \oplus z \oplus i)\} = \Gamma_i(S).$$

(3) The remaining possibility is that the vertex is labeled $0 \dots 0xzW$ for some nonempty string of characters W . It is only adjacent to the labels corresponding to all changes of the leading character in the label, and also to the label $0 \dots 0x(x \oplus z)W$. Clearly, applying Γ_i (changing the last character in W) preserves these adjacencies.

Thus all edges are preserved. \square

Theorem 6.11. *This recursive labeling is the same labeling as described in Theorem 6.8.*

Proof. Since Lemma 6.10 shows that applying some Γ_i to a subgraph does not change the edges within that subgraph, we only need show that the new edges drawn between the subgraphs are in fact the correct connections. Note that Γ_0 is the identity permutation since we do not permute the 0^{th} subgraph.

So fix some subgraph i in L_d^n , and take the j^{th} corner vertex where $j \neq i$. Recall that Γ_i was first applied to the subgraph and then i was appended. Thus, since it is a corner, the label has the form $0 \dots 0(j \oplus i)i = v$.

Now we claimed in our construction method that this vertex should connect to the i^{th} corner of j^{th} subgraph. Similar to above, the j^{th} subgraph had Γ_j applied and j appended, so the label is of the form $0 \dots 0(i \oplus j)j$. But this configuration may move to $0 \dots 0(i \oplus j)(i \oplus j \oplus j) = 0 \dots 0(i \oplus j)i = v$, since \oplus is commutative. Thus the two vertices are indeed adjacent. Since i, j were arbitrary, all connections between the subgraphs are correct, and we have proven the theorem. \square

6.4. Codeword Recognition. Using the G-U construction as described earlier in the paper, we know exactly what vertices are code vertices for each graph. Since these sets are defined recursively, we can use them in combination with the recursive labeling to produce recursive definitions for the actual codewords themselves. It will be helpful to refer to a labeled graph L_d^n with a code G_d^n or U_d^n associated with it. Thus we define C_d^n as the labeled graph L_d^n with the G_d^n code scheme, and H_d^n as L_d^n with the U_d^n code scheme. (We use this notation because the code vertices of C_d^n are the codewords of the PIECC on L_d^n , while the code vertices of H_d^n are a helper set of vertices.) Then when we write $\Gamma_i(C_d^n)$ or of $\Gamma_i(H_d^n)$, we mean that it permutes the labels “underneath” the code scheme, without moving the code vertices of the G_d^n or U_d^n graph. We also use the \bullet to denote an appending; $x \bullet i$ means append i to x .

Lemma 6.12.

For an odd iteration n , $C_d^n = C_d^{n-1} \bullet 0 \cup \bigcup_{i=1}^{d-1} \Gamma_i(H_d^{n-1}) \bullet i$.

For an even iteration n , $H_d^n = H_d^{n-1} \bullet 0 \cup \bigcup_{i=1}^{d-1} \Gamma_i(C_d^{n-1}) \bullet i$.

Proof. This follows immediately from the G-U construction and the recursive labeling. For odd n , the 0^{th} subgraph of the labeled graph C_d^n is formed by placing a copy of L_d^{n-1} and appending 0. The G-U construction shows that the 0^{th} subgraph is a G graph, hence the $C_d^{n-1} \bullet 0$. For all other i ,

the i^{th} subgraph was labeled using $\Gamma_i(L_d^{n-1})$, and the code vertices chosen using the U graph. Thus the terms $\Gamma_i(H_d^{n-1})$. The other equation is obtained similarly. \square

Recall that in the G-U construction, we often have to rotate some G or U scheme so that the top vertex becomes some other corner vertex. This plays a large role in our codeword definitions, so we formalize that rotation here. Let R_i be the permutation on the graphs G_d^n and U_d^n that rotates the entire set $2\pi i/d$ counterclockwise; that is it rotates so that the top corner, formerly at the 0^{th} corner, now lies at the i^{th} corner. See Figure 21 for an example. Note that the function R_i only moves the code vertices, *not* the labels associated with them, so that if we write $R_i(C_d^n)$ or $R_i(H_d^n)$, we mean that R_i rotates the code scheme “above” the labels without moving or changing them.

FIGURE 21. An example of the R function

Theorem 6.13.

- (1) For n odd, $R_i(C_d^n) = \Gamma_i(C_d^n)$.
- (2) For n even, $R_i(H_d^n) = \Gamma_i(H_d^n)$.
- (3) For n even, $\Gamma_i(C_d^n) = C_d^n$.
- (4) For n odd, $\Gamma_i(H_d^n) = H_d^n$.

Proof. These four results are so closely linked that the truth of any one of them depends on the truth of at least one other. We prove them using as little dependency as possible, and at the end show that we know enough base cases to establish each result for all iterations.

Beginning with (1), take some C_d^{2m+1} and consider what happens when we apply R_i . Recall that the G graph is made up of one smaller G graph and $d - 1$ smaller U graphs. First, the i^{th} subgraph, which had Γ_i applied to it in the recursive labeling and all of whose labels end in i , has become the one G subgraph. This gives us $\Gamma_i(C_d^{2m}) \bullet i$.

Next, for $j \neq i$, the j^{th} subgraph, which had Γ_j applied to it in the recursive labeling and all of whose labels end in j , is a U graph. However, this U graph has been rotated so that its top vertex is its i^{th} corner vertex, giving us $R_i \circ \Gamma_j(H_d^{2m}) \bullet j$ (where the \circ denotes normal function composition).

Thus in total we have that $R_i(C_d^{2m+1}) = \Gamma_i(C_d^{2m}) \bullet i \cup \bigcup_{j \neq i} R_i \circ \Gamma_j(H_d^{2m}) \bullet j$. But suppose (3) held; then the first element would be just $C_d^{2m} \bullet i$. Supposing (2) holds allows us to rewrite the union as $\bigcup_{j \neq i} \Gamma_i \circ \Gamma_j(H_d^{2m}) \bullet j = \bigcup_{j \neq i} \Gamma_{i \oplus j}(H_d^{2m}) \bullet j$.

Now applying Γ_i to both sides, we would have

$$\Gamma_i \circ R_i(C_d^{2m+1}) = \Gamma_i(C_d^{2m} \bullet i) \cup \Gamma_i \left(\bigcup_{j \neq i} \Gamma_{i \oplus j}(H_d^{2m}) \bullet j \right) = C_d^{2m} \bullet 0 \cup \bigcup_{j \neq i} \Gamma_{i \oplus j}(H_d^{2m}) \bullet (j \oplus i).$$

Since $j \neq i$, we know $j \oplus i = i \oplus j$ is not 0, but can be any other character in $\{1, \dots, d-1\}$. Thus

$$\Gamma_i \circ R_i(C_d^{2m+1}) = C_d^{2m} \bullet 0 \cup \bigcup_{j \neq i} \Gamma_{i \oplus j}(H_d^{2m}) \bullet (i \oplus j) = C_d^{2m+1}$$

by Lemma 6.12.

Thus, when their domains are restricted to odd iterations of C_d^n , we have that $\Gamma_i \circ R_i = I$, so that $R_i = \Gamma_i^{-1} = \Gamma_i$ since numbers are their own inverses under bitwise addition. Therefore we've shown (1) if (2) and (3) are true.

Now the argument for (2) is only notationally different from above and gives that, when restricted to even iterations of H_d^n , we get that

$$\Gamma_i \circ R_i(H_d^{2m}) = H_d^{2m-1} \bullet 0 \cup \bigcup_{j \neq i} \Gamma_{i \oplus j}(C_d^{2m-1}) \bullet (j \oplus i) = H_d^{2m}$$

by Lemma 6.12 if (1) and (4) are true. Thus we again have that restricted to this domain $R_i = \Gamma_i$.

For (3), take some C_d^{2m} and consider its construction. In the labeling the top (0^{th}) subgraph was not permuted, and then the G_d^{2m-1} graph was placed on top, simply giving us $C_d^{2m-1} \bullet 0$.

Now for any other subgraph i , we know Γ_i was applied. Also, however, the G_d^{2m-1} graph was rotated, moving its top corner to the i^{th} corner; that is, R_i was also applied. But if (1) holds, then over the domain of C_d^{2m-1} , we know $R_i = \Gamma_i = \Gamma_i^{-1}$, so the two permutations cancel, giving us only $C_d^{2m-1} \bullet i$. Thus if we would now apply any Γ_j , all we would do is change the last characters, which has no effect on the set of codewords $\bigcup_{i=0}^{d-1} C_d^{2m-1} \bullet i$ (it only changes their order on the graph).

The argument for (4) is the same as for (3); H_d^{2m+1} is constructed with an unpermuted 0^{th} subgraph, giving us $H_d^{2m} \bullet 0$. For other subgraphs i , both Γ_i and R_i were applied, so that if (2) holds, we have the set of codewords $\bigcup_{i=0}^{d-1} H_d^{2m} \bullet i$, which is unchanged under any Γ_j .

Finally, we show that we have a sufficient base to prove these claims for all iterations. First, we know that (4) holds for H_d^1 —no Γ_j can have an effect here, since it is the empty set.

Next, we establish (1) for C_d^1 , whose sole codeword is 0. We know that Γ_i sends i to 0, making i the codeword. Similarly, R_i rotates the G graph from the top to the i^{th} corner, again leaving i as the codeword. Thus the two functions are the same here.

Now, combining these two cases, we are able to prove (2) for H_d^2 . Also, knowing that (1) holds for C_d^1 establishes (3) for C_d^2 . These cases are sufficient to show each of these claims for all n . \square

Corollary 6.14. *For any dimension $d = 2^m$, the codewords for the Powers of Two labeling are described by the following recursive definitions:
for an even iteration n ,*

$$\begin{aligned}
 C_d^n &= \bigcup_{i=0}^{d-1} C_d^{n-1} \bullet i \text{ and} \\
 H_d^n &= H_d^{n-1} \bullet 0 \cup \bigcup_{i=1}^{d-1} \Gamma_i(C_d^{n-1}) \bullet i \\
 \text{for an odd iteration } n, \\
 C_d^n &= C_d^{n-1} \bullet 0 \cup \bigcup_{i=1}^{d-1} \Gamma_i(H_d^{n-1}) \bullet i \text{ and} \\
 H_d^n &= \bigcup_{i=0}^{d-1} H_d^{n-1} \bullet i
 \end{aligned}$$

Proof. The middle two equations are Lemma 6.12. The other two were proved within parts (3) and (4) of Theorem 6.13. □

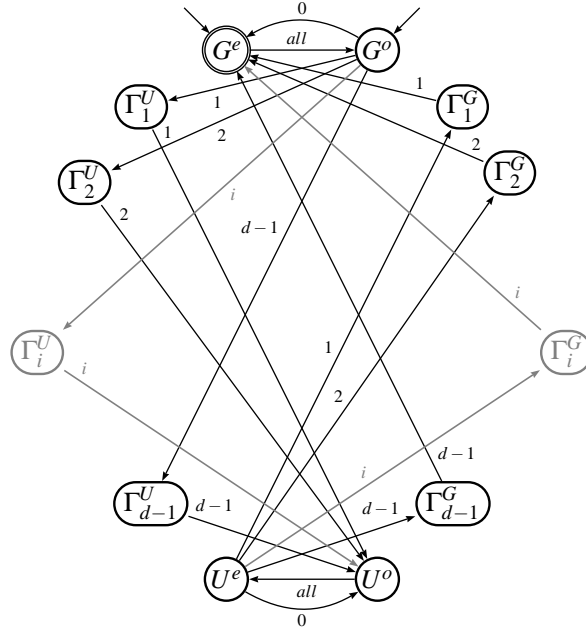


FIGURE 22. Nondeterministic finite-state Codeword Recognizer for the Dimension 2^m Puzzle

Corollary 6.15. *A nondeterministic finite-state machine (Figure 22) that reads strings right to left and recognizes codewords follows directly from the recursive definitions in corollary 6.14.*

Since the recognizer has only one accepting state, we can easily reverse the machine to make it deterministic by making it read the strings in the opposite direction, left to right. We show this deterministic machine in Figure 23. (For theory on finite-state machines, see [21].) We leave out several edges between the various Γ_i states so the structure of the machine can be more easily seen; these edges can be easily added by following those Γ_i permutations directly. All edges involving the G states and U states are shown. Also note that the machines for the labels of dimensions 2 and 4 are of this form.

6.5. Error Correction.

Theorem 6.16. *The finite-state machine shown in Figure 24 correctly error-corrects labels. The recognizer must first be used; the ending state if the recognizer is the start state for the corrector. Note that for the transitions labeled with a Γ permutation, each character follows that arrow and permutes according to that Γ . All other edges do not change the characters.*

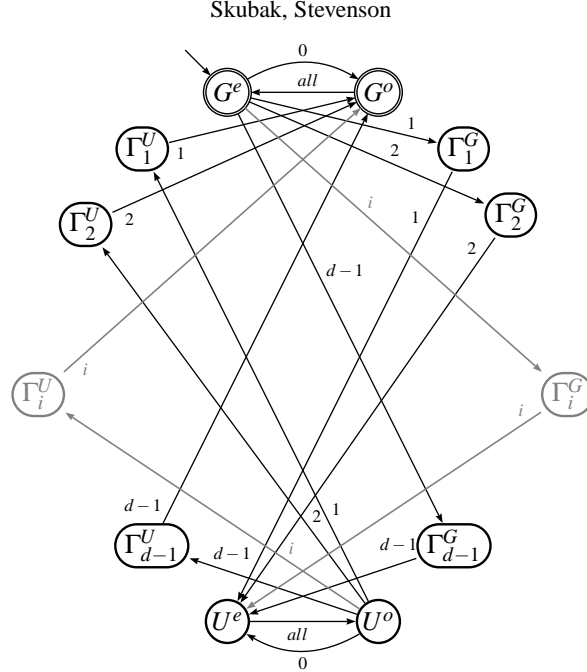


FIGURE 23. Deterministic finite-state Codeword Recognizer for the Dimension 2^m Puzzle

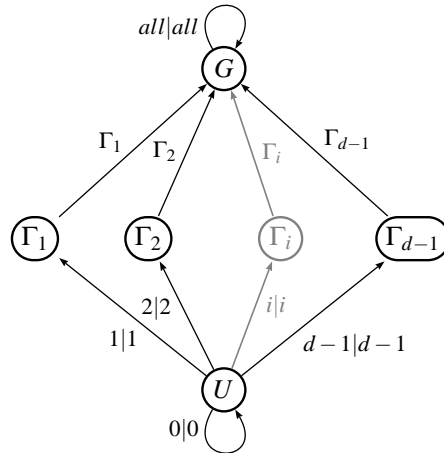


FIGURE 24. finite-state Error Corrector for the Dimension 2^m Puzzle

Proof. If we use the nondeterministic codeword recognizer in Figure 22, by a parity argument we find that for any word, we end in one of $d + 1$ states: the codeword G^e state, the $d - 1$ states Γ_i^U , and the U^e state. Thus this must be sufficient information to correct. If a word ends in the G state, clearly we want to make no change, which is clearly reflected in the corrector. Now recall that this recognizer reads right to left, so that the leftmost bit is the last read. Then if a word ends in some Γ_i^U state, we can retrace one step, analogous to returning one bit to the right. Note that this will always place us in either the G_0 state or a Γ_i^G state. Each of these states has exactly one 1-step path to the accepting state; thus we need only change the last bit to the bit labeling this path to make our word a codeword. This corresponds to the $d - 1$ middle states in the error-corrector; since the corrector runs left to right, if we begin in one of these states, we change only the leftmost bit to the required character, which is dictated by Γ_i . This is clearly the case when the word is in the same K_d^1 complete graph as a codeword and only needs change its leftmost bit, or 0^{th} piece.

Now, the more complicated case is correcting those words which end the recognizing process in the U^e state. This occurs when a word must change to a word outside of its immediate subgraph, i.e. piece $j \neq 0$ must change orientations to $f(j-1) \oplus f(j)$ according to the rule. Then we must have the 0^{th} through $(j-2)^{\text{nd}}$ pieces at the 0 orientation, so we want to disregard all initial zeros. Clearly the error-correcting machine, beginning at the U state, does this. Then the $(j-1)^{\text{st}}$ piece should not be changed, but is important in deciding what the j^{th} piece may change to. This also is reflected in the corrector; the first nonzero orientation is not changed, but takes the machine to a state that remembers $f(j-1)$. Since we want $f(j)$ to change to $f(j) \oplus f(j-1)$, we want to use $\Gamma_{f(j-1)}$, which the machine does. These are all the possible cases, therefore the machine is able to properly correct words. \square

7. OTHER EVEN DIMENSIONS

Together with the SF labeling, we have now classified recursive labelings corresponding to puzzles on the families of iterated complete graphs of odd dimension and of dimensions that are powers of two. The SF Puzzle is a Towers-of-Hanoi-like puzzle, while the latter puzzles, based on the Spin-Out puzzle, are completely different. What about the other even dimensions? It is clear that these two types of puzzle do not easily extend to these other dimensions, the first of which is 6. Instead we can combine the two types of puzzles to produce the puzzle and the labeling that we're looking for.

7.1. **Dimension 6.** The smaller example of the combined puzzle is for dimension $d = 3 \cdot 2^1$. As in the Towers of Hanoi, we have n pieces stacked on three towers labeled 0, 1 and 2 from left to right. Usually, our pieces are disks that have no orientation, so that only a piece's tower matters. Now, we will give each piece two orientations, 0 and 1, which correspond to a horizontal and a vertical orientation respectively (which is why we change the terminology from 'disk'). For a given piece j , we will combine the tower t_j and orientation r_j to define its **total orientation** $f(j)$ as $t_j \cdot 2 + r_j$. Therefore there are six total orientations that each piece can be in. The distinct combinations of towers and relative orientation clearly give distinct values from 0 to 5. This is illustrated in Figure 25.

FIGURE 25. Plate positions for the Dimension 6 Puzzle

We label configurations of the puzzle, as usual, 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. Thus the game configuration for iteration $n = 2$ in Figure 26 corresponds to the label 50.

FIGURE 26. An Example Configuration of the Dimension 6 Puzzle

7.1.1. *Rules.* The normal Towers of Hanoi rules still apply. Plates must always be stacked from largest on the bottom to smallest at the top. Only the smallest plate on a tower may be moved, and it may only move to a tower containing either no plates or only plates larger than itself. (Note

that this requires all the plates smaller than this plate to be together on one tower.) As usual, the smallest piece may always move in any way, to any orientation.

In addition, piece $j \neq 0$ piece may only be moved if

- (1) the $(j - 2)^{\text{nd}}$ piece through the 0^{th} piece are all the same and have orientation 0 (they are equivalent to $0 \pmod{2}$)
- (2) t_{j-1} equals t_0 through t_{j-2}
- (3) if $t_j = t_{j-1}$, then $r_{j-1} \neq 0$

Note that this means that even if we can move a piece in regular Towers of Hanoi, we may not be able to move it in this new puzzle.

If the conditions above are met, then piece j may move to tower $2t_{j-1} - t_j$ at the same time as it moves to orientation $r_{j-1} \oplus r_j$. A few examples of moves can be found in Figure 27.

FIGURE 27. Example moves for the Puzzle on Dimension 6

7.1.2. *Graph and Label.* These rules give us the labeling of the graph shown in Figure 28. Note that there are two Towers of Hanoi labelings embedded in each graph, and that there are three reflected binary Gray code labelings on three of the outside edges (each with some simple permutations of characters).

FIGURE 28. The Puzzle Labeling for K_6^1 and K_6^2

7.2. General Dimensions. Every even number can be written as $q \cdot 2^m$ where q is odd and $m \geq 1$. So, as we did for dimension 6, we will combine the two styles of puzzle, an SF puzzle of dimension q and an extended Spin-Out puzzle with dimension 2^m , to define our 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. Since this puzzle can be represented by K_d^n as we will prove, we know that the minimum solution path for any two configurations is not longer than $2^m - 1$, the diameter of the graph. Another way of “solving” this puzzle might be to choose two configurations and move from one to the other. Again, the focus of this paper is the puzzle’s relationship to the family of iterated complete graphs.

Definition 7.1. *For a dimension $d = q \cdot 2^m$ with q odd, define the puzzle as follows. The puzzle has q towers, numbered 0 through q . There are n puzzle pieces, each consisting of m spinners. Each piece has 2^m orientations, numbered 0 through $2^m - 1$, on each of q towers. These orientations are defined exactly as in the extended Spin-Out puzzle (see Section 6), by writing the orientation in binary and letting each bit represent one spinner of the piece.*

*For a given piece j , the combination of the tower t_j and orientation r_j defines the **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.*

Pieces are numbered from the 0th piece, the least restricted (which can be thought of as the smallest or leftmost piece), through the $(n - 1)$ st piece, the most restricted (biggest or rightmost). There are three rules that define legal moves between configurations:

- (1) **The 0th Piece Rule** *The 0th 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 $0 \pmod{2^m}$; 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*

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

at the same time as its orientation changes to

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

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.

Lemma 7.2 (Reversibility). *All moves are reversible and all edges are undirected.*

Proof. First note that all moves by the 0th piece are reversible because that piece can always move anywhere.

So suppose a piece $j \neq 0$ can move. Let $x \equiv 0 \pmod{2}$. Also let $0 \leq w, y, z < d - 1$ and W be the empty string or composed of characters from 0 through $d - 1$. Then this puzzle configuration has the form $x \dots xywW$. Suppose it can move to $x \dots xyzW$ (where there may be any nonnegative number of leading x characters.) Let the tower of the piece at total orientation z be called t_z and the orientation of this piece be r_z . Similarly define t_y , r_y , t_x , and r_x . Then t_z is $2t_y - t_w \pmod q$, and $r_z = r_y \oplus r_w$.

So consider the configuration $x \dots xyzW$. The piece at orientation z may move to a tower $(2t_y - t_z) \pmod q = (2t_y - (2t_y - t_w)) \pmod q = t_w$. It may change to the relative orientation $r_y \oplus r_z = r_y \oplus (r_y \oplus r_w) = r_w$ by the associativity of \oplus . Since the tower and relative orientation completely define the total orientation, z may change to w and then $x \dots xyzW$ may move to $x \dots xywW$ \square

Lemma 7.3 (Corner Labels). *The configuration $x \dots x(x + y)$ where $x \equiv 0 \pmod{2^m}$ and $0 \leq y < 2^m$ has exactly $d - 1$ possible moves. In other words, it corresponds to a corner vertex.*

Proof. First note that any piece in this configuration with total orientation x cannot satisfy condition (c) and so cannot move. So consider the piece j with total orientation $x + y$. Since $y < 2^m$ and $f(j - 1) = x$, we know that $t_j = t_{j-1}$. Also, we have that $r_{j-1} = 0$. Again by condition (c) this piece cannot move either. \square

Lemma 7.4 (Non-corner Labels). *All other configurations have exactly d possible moves, and so are non-corner vertices.*

Proof. We split the other vertices into three cases.

- (1) If a configuration begins in $x \dots x$ for $x \equiv 0 \pmod{2^m}$, then there must be a next character z that is on a different tower, or it would be a corner by Lemma 7.3. Then condition (c) does not apply. Clearly conditions (a) and (b) are also satisfied.

- (2) If a configuration begins in $x \dots x(x+y)$ for $x \equiv 0 \pmod{2^m}$ and $0 \leq y < 2^m$, then again there must be a next character z or it would be a corner by Lemma 7.3. Also assume that $y \neq 0$ since that is the case above. Then consider the piece with total orientation z the j^{th} piece. Clearly conditions (a) and (b) are satisfied, but since $t_j = t_{j-1}$, we must check that (c) is satisfied. But $r_{j-1} = y \neq 0$, and so we've proven this case.
- (3) Finally, if a configuration is neither a corner nor one of the above cases, it must begin with some zw with z not equal to $0 \pmod{2^m}$; that is, $r_0 \neq 0$. Conditions (a) and (b) are trivially satisfied, and since $r_0 \neq 0$, (c) is satisfied whether $t_j = t_{j-1}$ or not.

□

Lemma 7.5. *When $f(j-1) = x+y$ for $x \equiv 0 \pmod{2^m}$ and $0 \leq y < 2^m$, the function determining the relative orientation of a piece j , defined by $r_{j-1} \oplus r_j = y \oplus r_j$, attains all values 0 through $2^m - 1$, and equals r_j if and only if $y = 0$.*

In terms of the puzzle, we take a piece j with a given total orientation. Then we keep t_{j-1} constant but change r_{j-1} to all possible values. Then the range of relative orientations to which piece j may move has all possible values 0 through $2^m - 1$. Furthermore, piece j retains its orientation r_j if and only if $y = 0$ (thus if it does not change tower, this is not an actual move).

Proof. This proof is nearly trivial. Since r_j is fixed and y changes and both are less than $2^m - 1$, by the definition of \oplus , the bitwise addition $y \oplus r_j$ must attain all possible values 0 through $2^m - 1$. Also, clearly r_j is returned only if $y = 0$. □

Theorem 7.6. *The puzzle defined in Definition 7.1 for dimension d and iteration n corresponds to the iterated complete graph K_d^n . That is, each puzzle configuration represents a vertex and each legal move between configurations represents an edge.*

Proof. The 0^{th} piece, by the first rule, may always move to any other total orientation 1 through $d - 1$. Then the puzzle with only one piece has these d configurations and can move freely between them. This clearly corresponds to the complete graph $K_d = K_d^1$.

So assume that the puzzle with $n - 1$ pieces corresponds to K_d^{n-1} . We will add a new piece that is more restricted (or bigger, or to the right) than all the other pieces. Note that, by our rules, legal moves by the 0^{th} and by a j^{th} piece are not affected by the orientation or even existence of any pieces $j + 1$ or higher. Thus the new puzzle is simply d copies of the puzzle with $n - 1$ pieces. In each copy, the new, added piece has a different orientation 0 through $d - 1$. By our induction hypothesis, these smaller puzzles correspond to K_d^{n-1} subgraphs, so we need only show that they connect in the correct manner to create K_d^n .

Fix some i^{th} subgraph K_d^{n-1} . First we show that this subgraph has exactly one corner that is also a corner of K_d^n . Then we prove that the other $d - 1$ corners of the subgraph connect to each of the $d - 1$ other subgraphs, and that there are no other connections between them. (By Lemma 7.2, all edges are undirected as desired.) Then, since i is arbitrary, we will have proven our claim.

In the i^{th} subgraph, each label was lengthened by the character i . By Lemma 7.3, the corner of K_d^n in this subgraph must then have the label $x \dots xi$ with $xequiv 0 \pmod{2^m}$ and $i = x + y$ with $0 \leq y < 2^m$. But since $y < 2^m$, we have only one choice for x so that $i = x + y$. Then the vertex $x \dots xi$ is the one and only corner of K_d^n in this subgraph.

Now consider some other corner vertex in the i^{th} subgraph of K_d^{n-1} . Then by Lemma 7.3 its label in K_d^n must have the form $w \dots w(w+z)i$ with $w \equiv 0 \pmod{2^m}$ and $0 \leq z < 2^m$. Also we do not

have that $z = 0$ and $w = x$ at the same time, or this configuration is the corner we have already addressed. Denote the piece with total orientation piece j . We have two cases:

- (1) If $w = x$, then $z \neq 0$. Therefore $w \dots w(w+z)i$ does not have the correct form to be a corner of K_d^n . Since it was, however, a corner of K_d^{n-1} , we must be able to change i . Note that $t_j = t_{j-1}$ since $w = x$ and $i = x + y$. Thus the last piece stays on this tower since $2t_{j-1} - t_j = t_j$. By Lemma 7.5, for $1 \leq z < 2^m$, we have that $r_{j-1} \oplus r_j$ becomes everything except r_j (since the tower remains the same, that value would represent no actual change of the piece). Then these corners connect to $2^m - 1$ other subgraphs.
- (2) On the other hand, suppose $w \neq x$. Then $t_j \neq t_{j-1}$ since x and w represent total orientation both equivalent to $0 \pmod{2^m}$ but distinct. By Lemma 7.5, for each value of w , the value of r_j changes to all of its possible 2^m values; there are $q - 1$ possible values for w (since w is not x). Also, since our total orientation is $w + r_j$ and $r_j < 2^m$, all these values are distinct. Thus these corners of the subgraph connect to exactly $(q - 1)2^m$ other subgraphs. Note also that these subgraphs must be different from those in (1) since those are on a different tower. Then we have a total of $2^m - 1 + (q - 1)2^m = q \cdot 2^m - 1 = d - 1$ connections to distinct other subgraphs as needed.

Finally, if we have some vertex that was not a corner in K_d^{n-1} , then it has degree d by Lemma 7.4. We know, also by this lemma, that no vertex can have degree higher than d . Since adding a more restricted piece cannot change legal moves by pieces less restricted than itself, we know that our new piece must not be able to move. Thus the connections between subgraphs above are the only such connections.

□

7.3. Conclusions on the Puzzle for General Dimension. As desired, we have now found a puzzle that can represent the iterated complete graph K_d^n for any iteration n and any dimension d . The simplest cases, $d = 2$ and $d = 3$, have been known and well-studied for years. (The Towers of Hanoi puzzle supposedly dates back to Buddhist legend.) Extensions to these two puzzles provide interesting puzzles in their own right, but we show that we can go further by combining the two types using a simple and beautiful theorem: every integer is an odd number times a power of 2.

Further research could be done to prove that the puzzle for general dimensions has finite-state machines for codeword recognition and error-correction. We believe this to be the case, though the recursive definitions quickly become very complicated. Also, for both Spin-Out and the Towers of Hanoi puzzle, we have an easy way to map the codewords to a subset of the natural numbers and back, processes that are called decoding and encoding. Even for the dimension 4 Reflection Puzzle these processes are not simple though, again, we believe they exist.

Even with these few open areas, this general puzzle makes complete a rather intriguing relationship between a series of puzzles and the family of iterated complete graphs.

8. NEW LABELINGS

The following are a couple of labelings which deserve to be included in this discussion. It is surprising that they were not presented in any past research. While they do not satisfy all of the properties that we would like, they are intuitive and have some nice characteristics.

8.1. The Corner-Distance Labeling. Recall that the corners of K_d^n are the d vertices whose degrees are $d - 1$. Assign each corner of K_d^n a unique number from $\{0, \dots, d - 1\}$. It makes sense to assign 0 to the “top” corner and move in one direction, for example counterclockwise, numbering sequentially. We will use this convention. We give each vertex the label $\delta(0), \dots, \delta(d - 1)$, where $\delta(i)$ is the distance from the vertex to corner i . Figure 29 has some examples.

FIGURE 29. Corner-distance labels on K_3^1 , K_3^2 , and K_3^3 .

Now define a function

$$l : L_d^n \times \{0, \dots, d - 1\} \mapsto L_d^n \quad \text{by}$$

$$l(\delta, i) = \delta(0) + 2^n, \dots, \delta(i - 1) + 2^n, \delta(i), \delta(i + 1) + 2^n, \dots, \delta(d - 1) + 2^n.$$

That is, given a label δ and the position i of one of its components, l adds 2^n to all but the i^{th} component of that label. The function l also operates on sets, mapping all labels in the set without changing their relative positions. The value of n will be clear when we use this notation.

Example 8.1. Given that $L_2^1 = 01 - 10$, then

$$L_2^2 = l(L_2^1, 0) - l(L_2^1, 1) = 03 - 12 - 21 - 30.$$

Iterating from L_d^n to L_d^{n+1} involves connecting d copies of L_d^n . The vertices in the i^{th} subgraph of K_d^n will remain the same distance from the i^{th} corner, but will be 2^n further from the other corners. Thus, L_d^n satisfies the recursion in Figure 30, using $d = 4$ as an example.

FIGURE 30. Recursive corner-distance label construction for dimension 4.

Theorem 8.2. *The corner-distance labeling assigns a unique label to each vertex of K_d^n .*

Proof. Fix $d > 2$. K_d^n is the complete graph on d vertices. Each vertex of K_d^n has degree $d - 1$ and so is a corner vertex. The labels are strings of length d over $\{0, 1\}$ with exactly one zero. Only corner i 's label has a zero in the i^{th} position because no other vertex is distance zero from corner i . Thus, no labels on K_d^1 are repeated. Now assume that no labels on K_d^n are repeated for some $n > 1$. Given two labels δ_1 and δ_2 of vertices on K_d^n , we have $\delta_1(j) \neq \delta_2(j)$ for some j by our assumption. When δ_1 and δ_2 are in the same subgraph of K_d^n , iterating to K_d^{n+1} involves adding 2^n to the same component of δ_1 and δ_2 . Of course, this preserves the inequality. When δ_1 and δ_2 are in different subgraphs, iterating involves adding 2^n to different components. Then we must check that $\delta_1(j) \neq \delta_2(j) \pm 2^n$. This is true since $\delta_1(j), \delta_2(j) \in \{0, \dots, 2^n - 1\}$. Thus, every pair of labels δ'_1 and δ'_2 of K_d^{n+1} will have $\delta'_1(j) \neq \delta'_2(j)$ for some j . Therefore, the corner-distance labeling assigns a unique label to each vertex. \square

The corner-distance labels can be used to find the distance from one vertex to another. First, we introduce some notation. If v is a vertex, $v(i)$ means the i^{th} component of the label of v . Let $\min(v)$ represent the position of the smallest component of the label of v . Remember that the first component is in the 0^{th} position.

Example 8.3. *If the label of v is 671, then $\min(v) = 2$.*

The following algorithm finds the distance d from vertex x to vertex y on K_d^n .

Algorithm 8.4. *If $\min(x) = \min(y) = i$, then decrement n by one and subtract 2^n from all but the i^{th} components of each label. Let these strings be the new labels for x and y . Repeat this until $\min(x) \neq \min(y)$ or $n = 0$. (If the original labels of x and y have $\min(x) \neq \min(y)$, then we have not done anything yet.) If $n > 1$, decrement n once more and subtract 2^n from all but the $\min(x)^{\text{th}}$ label of x and all but the $\min(y)^{\text{th}}$ label of y .*

The distance d from x to y is given by

$$d(x, y) = \begin{cases} n & n = 0, 1 \\ \min \left\{ \begin{array}{l} x(\min(y)) + y(\min(x)) + 1, \\ x(q) + y(q) + 2^{n-1} + 1 : q \neq \min(x), \min(y) \end{array} \right\} & n > 1 \end{cases} .$$

Proposition 8.5. *Algorithm 8.4 correctly finds the distance between two vertices.*

Proof. When $\min(x) = \min(y)$, x and y are in the same K_d^{n-1} subgraph. Then, by the recursive structure of the corner-distance labeling, subtracting 2^{n-1} from all but the smallest components of the labels of x and y gives their corner-distance labels on K_d^{n-1} . Continuing this gives the corner-distance labels of x and y on the smallest subgraph that contains x and y . If $n = 1$, the labels have been reduced to those on the complete graph K_d^1 . Then $d(x, y) = 1$. If $n = 0$, the labels of x and y on K_d^1 have their minimum in the same position. These labels are strings over $\{0, 1\}$ with exactly one zero. Then $x = y$ and $d(x, y) = 0$. If $n > 1$, then x and y are in K_d^{n-1} subgraphs of K_d^n . A path from x to y will go through corners of both subgraphs. Then subtracting 2^{n-1} from all but the $\min(x)^{\text{th}}$ label of x and all but the $\min(y)^{\text{th}}$ label of y gives their labels on K_d^{n-1} . Now $x(i)$ and $y(i)$ give the distances to the i^{th} corners of their subgraphs. The shortest path from x to y goes directly from x 's

subgraph to y 's subgraph or else passes through one other subgraph. In the former case, we go from x to the corner of x 's subgraph adjacent to y 's subgraph, one across to y 's subgraph, and then from the corner of y 's subgraph adjacent to x 's subgraph to y . That is $\min\{x(\min(y)) + 1 + y(\min(x))\}$. In the latter case, we go from x to a subgraph corner q which is neither adjacent to y 's subgraph nor a corner of K_d^n . Then one across to the other subgraph q , of which the distance from corner to corner is $2^{n-1} - 1$. Then one across to y 's subgraph and the distance from corner q of y 's subgraph to y . That is $x(q) + 1 + (2^{n-1} - 1) + 1 + y(q)$.

The distance along a path through two subgraphs is at least $2(2^{n-1} - 1) + 3 = 2^n + 1$, while the direct path has $x(\min(y)) + y(\min(x)) + 1 \leq 2(2^{n-1} - 1) + 1 = 2^n - 1$. Thus, paths through two or more subgraphs need not be considered. \square

Proposition 8.6. *The set of vertices on K_d^n is a metric space.*

Proof. We will show that the distance d given by Algorithm 8.4 is a metric on the set of vertices on K_d^n . Exchanging x and y in the algorithm yields the same distance. This is trivial to check. Then $d(x,y) = d(y,x)$. We established in the proof of Proposition 8.5 that $d(x,y) = 0 \Rightarrow x = y$. If $x = y$, then $\min(x) = \min(y)$ for every step in the algorithm so we will decrement to $n = 0$. Then $x = y \Rightarrow d(x,y) = 0$.

Only left is the triangle inequality. Distance is minimal by definition. Proposition 8.5 establishes that $d(x,z)$ is the distance traveled on a shortest path from x to z . Assume for some vertex y that $d(x,z) > d(x,y) + d(y,z)$. Then going through y is a shorter path from x to z , a contradiction since $d(x,z)$ is already minimal. Then it must be that $d(x,z) \leq d(x,y) + d(y,z)$. Equality holds only when y is on a shortest path from x to z . \square

We can now refer to the distance between two vertices, rather than from one to another, without confusion.

The corner-distance labeling on K_2^n has the finite-state codeword recognizer and error-corrector presented in Figure 8.1.

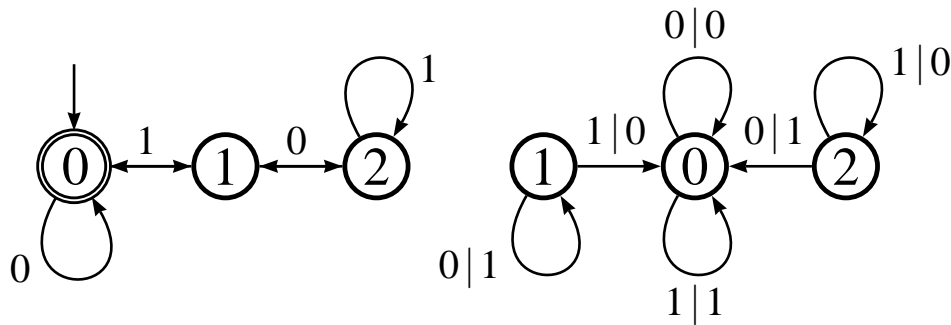


FIGURE 31. Machines for recognition of codewords (left) and error-correction (right) on K_2^n for the corner-distance labeling.

The recognizer reads from right to left the first component of a label in binary and checks if it's divisible by three. The double circle (state 0) is the accepting state. If you end up in state 1 or 2, start at that state in the error corrector. Read from right to left the first component of the label in binary, recording the output. The corrector has *input* | *output*. The output $\delta(0)$ is the corrected

first component of the label. The second component $\delta(1)$ has $\delta(1) = (2^n - 1) - \delta(0)$, which the reader may verify.

The machines in Figure 8.1 were given by Bode[5]. The recognizer ends in state r when a binary number is $r \bmod 3$. The corrector then “sends” the number to the nearest $0 \bmod 3$ number.

Conjecture 8.7. *The corner-distance labeling has no finite-state machine for codeword recognition for $d > 2$.*

A proof of Conjecture 8.7 would show that the set of codewords on K_d^n for a fixed $d > 2$ is not a regular language. This normally requires appealing to the Pumping Lemma. We were not successful. Even when the labels are written in base d , it seems that an indefinite amount of information from the first component would have to be remembered to distinguish between codeword and non-codeword. Of course, a finite-state machine can only deal with a finite amount of information.

There is, however, Algorithm 8.8 for codeword recognition on the corner-distance labeling.

Algorithm 8.8. *Given a label D_0, D_1, \dots, D_{d-1} on K_d^n , $G(D_0, D_1, \dots, D_{d-1}, n) = \text{true}$ if and only if D_0, D_1, \dots, D_{d-1} is a codeword.*

The base cases are given by

$$\begin{aligned} G(D_0, D_1, \dots, D_{d-1}, 1) &= \begin{cases} \text{true}, & D_0 = 0, D_{i \neq 0} = 1 \\ \text{false}, & \text{else} \end{cases} \\ U(D_0, D_1, \dots, D_{d-1}, 1) &= \text{false} \end{aligned}$$

The base cases are obvious because we require that the top vertex, labeled $01\dots$ be a codeword. The other cases are given by

- *when n is even*

$$\begin{aligned} G(D_0, D_1, \dots, D_{d-1}, n) &= G(D_i, D_{i+1} - 2^{n-1}, \dots, D_{i+d-1}, n-1), & D_i \text{ is min} \\ U(D_0, D_1, \dots, D_{d-1}, n) &= \begin{cases} U(D_0, D_1 - 2^{n-1}, \dots, D_{d-1} - 2^{n-1}, n-1), & D_0 \text{ is min} \\ G(D_0 - 2^{n-1}, \dots, D_i, \dots, D_{d-1} - 2^{n-1}, n-1), & D_{i \neq 0} \text{ is min} \end{cases} \end{aligned}$$

- *and when n is odd*

$$\begin{aligned} G(D_0, D_1, \dots, D_{d-1}, n) &= \begin{cases} G(D_0, D_1 - 2^{n-1}, \dots, D_{d-1} - 2^{n-1}, n-1), & D_0 \text{ is min} \\ U(D_0 - 2^{n-1}, \dots, D_i, \dots, D_{d-1} - 2^{n-1}, n-1), & D_{i \neq 0} \text{ is min} \end{cases} \\ U(D_0, D_1, \dots, D_{d-1}, n) &= U(D_i, D_{i+1} - 2^{n-1}, \dots, D_{i+d-1}, n-1), & D_i \text{ is min} \end{aligned}$$

where addition of subscripts is mod d .

Algorithm 8.8 follows directly from the G-U construction and the recursive structure on the corner-distance labeling. Permuting the components of a label, as G does when n is even and U

when n is odd, corresponds to rotating a subgraph. Recall that constructing G_d^n requires making d copies of G_d^{n-1} and connecting them so that the top vertex of every copy remains unconnected. So the k^{th} copy will rotate $2\pi k/d$ radians. Then corner 0 becomes corner k , 1 becomes corner $k+1$, etc. Corner $d-1$ becomes corner 0 of the first G_d^{n-1} subgraph. Since the corners change the components of the labels permute accordingly. This is where the addition mod d comes from. Subtracting 2^{n-1} from all but the minimum component of a label corresponds to pairing down to its subgraph. In a sense, this is the opposite of iterating.

8.2. The Subgraph Labeling. The G-U construction produces the PIECC on K_d^n by breaking it down into subgraphs, and subgraphs of subgraphs, etc. Then it should be easy to recognize codewords with a labeling scheme that corresponds to subgraphs. We were right about this. We will show that the Subgraph labeling supports finite-state recognition.

To construct the labeling, start with the complete graph K_d^1 on d vertices. Label each vertex of K_d^1 a unique number from $\{0, \dots, d-1\}$. It makes sense to assign 0 to the “top” corner and move in one direction, for example counterclockwise, numbering sequentially. We will use this convention. Let the vertex with label k be called the k^{th} vertex or vertex k . Iterating to K_d^2 , make d copies of K_d^1 and choose one to be the top. Form edges between the copies so that the top copy is adjacent to the top vertices of the others and vertex 0 of the top copy is still a corner. The top copy will be called the 0^{th} subgraph. Call a non-top copy the k^{th} subgraph, where k is vertex of the top copy to which the subgraph is adjacent. Now append a k to the labels in the k^{th} subgraph. Figure 32 has some examples.

FIGURE 32. Subgraph labels on K_3^1 , K_3^2 , and K_3^3 .

We see that label 221, for example, is first vertex in the second subgraph of the second subgraph. The labeling L_d^n satisfies the recursion in Figure 33, using $d = 4$ as an example.

Theorem 8.9. *The machine in Figure 33 correctly recognizes codewords of K_d^n with the subgraph labeling.*

Proof. This machine started out nondeterministic, with two start states and one accepting states. For the moment, imagine the roles of the start state and accepting states in Figure 33 are reversed. Note the four columns of states. The first column on the left (with one state) corresponds to G even, the second to G odd, third to U even, and fourth to U odd.

When n is even, we start at the top state in the G even column. Reading k we move to G odd and now we are in a subgraph whose k^{th} corner was a top vertex because G rotates when n is even.

FIGURE 33. Recursive subgraph label construction for dimension 4.

FIGURE 34. The codeword recognizer for subgraph labeling on K_d^n .

The d G odd states “remember” which of the d corners was top. Now, when n is odd, G makes its top subgraph G even and the others U even without rotating. When n is even, U makes its top subgraph U odd in column 4 and the others G odd. Note that the d states in column 3 also know which was top. When n is odd, U reads k and rotates a copy of U even so the “top” is at the k^{th} corner. This covers all the transitions.

In the G-U construction, only G_d^1 , corresponding to a G odd state, has a codevertex and it is a top vertex. The last component j of a subgraph label gives the location of the vertex on K_d^n . The label is a codeword if the label’s vertex is in a G_d^1 whose top was rotated to the j^{th} vertex. Thus, the only way to land on G even, the accepting state, is to be on a G odd state and read from the last component that the vertex started as a top vertex.

This proves the non-deterministic machine you were asked to imagine. Note that each transition is bidirectional. We could read the labels in reverse order. If we start at the accepting state, we will end at one of the two start states. So we exchange their roles to obtain the deterministic machine. \square

Theorem 8.10. *The machine in Figure 33 has a minimal number of states to recognize codewords on K_d^n .*

Proof. The state minimization algorithm[21] groups states that are equivalent to arrive at a minimal number of states. We begin by grouping states into accepting and non-accepting. If we start at any two states A and B in a group, read a string, and end up in states belonging to different groups, we know that A and B are distinct. In this way, we determined that the machine is minimal when $d = 3$. In particular, after reading all possible strings (0, 1, and 2) of length one, the states in columns 1 and 4 are found to be distinct. Since the machine corresponding to $d > 3$ has all the states and transitions of the $d = 3$ machine, we know the state minimization algorithm splits up the states in columns 1 and 4 for all $d \geq 3$. Going from a d machine to a $d + 1$ adds two new states to the bottom of the machine. Starting at one new state and reading d takes us to the state in column 1. The other state goes to column 4. Thus, the new states must be distinct. \square

8.3. Conclusions on the New Labelings. No attempt was made to fit a puzzle to the corner-distance or subgraph labelings. No simple methods are known for various tasks like error correction, encoding, and decoding on the labelings. They do not have the Gray code property. On the other hand, they have some attractive qualities like easy construction and recognition. Also, each labeling on a graph has symmetries. The fact that the labels describe attributes of the actual graph may make them useful for further study of iterated complete graphs.

REFERENCES

- [1] A. Barg. Some new NP-complete coding problems. *Problems of Information Transmission*, 30(3):44–49, 1994.
- [2] A. Barg. Complexity issues in coding theory. *Handbook of Coding Theory*, 1998.
- [3] E.R. Berlekamp, R.J. McEliece, and C.A van Tilborg. On the inherent intractability of certain coding problems. *IEEE Trans. Inf. Theory*, IT-24(3):384–386, May 1978.
- [4] N. Biggs. Perfect codes in graphs. *J. Combinatorial Theory(B)*, 15:289–296, 1973.
- [5] David Bode. Alternate Labelings for Graphs Representing Perfect-One-Error-Correcting Codes. 1998.
- [6] J. Bruck and M. Noar. The hardness of decoding linear codes with preprocessing. *IEEE Trans. Inf. Theory*, IT-36:331–335, 1990.
- [7] P. Cull and E.F. Ecklund Jr. Towers of Hanoi and Analysis of Algorithms. *American Mathematical Monthly*, 92(6):407–420, June-July 1985.
- [8] Paul Cull, Mary Flahive, and Robby Robson. *Difference Equations*. Spring Science+Business Media, New York, 2005.
- [9] Paul Cull and Ingrid Nelson. Error-correcting codes on the towers of Hanoi graphs. *Discrete Math.*, 208/209:157–175, 1999.
- [10] Paul Cull and Ingrid Nelson. Perfect Codes, NP-Completeness, and Towers of Hanoi Graphs. *Bull. Inst. Combin. Appl.*, 26:13–38, 1999.
- [11] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.
- [12] R. Hamming. Error detecting and error correcting codes. *Bell Syst. Tech. J.*, 29:147–160, 1950.
- [13] R. Hill. *A First Course in Coding Theory*. Oxford University Press, Oxford, 1986.
- [14] Kathleen King. A new puzzle based on the SF labelling of iterated complete graphs. 2004.
- [15] Sandi Klavzar, Uros Milutinovic, and Ciril Petr. 1-perfect codes in Sierpinski graphs. *Bull. Austral. Math. Soc.*, 66:369–384, 2002.
- [16] Stephanie Kleven. Perfect Codes on Odd Dimension Serpinski Graphs. 2003.
- [17] J. Kratochvii. Perfect Codes in Graphs. *Proc. VII Hungarian Colloq. Combin. Eger*, 1987.
- [18] J. Kratochvii. *Perfect Codes in General Graphs*. Academia, Prague, 1991.
- [19] J. Kratochvii. Regular Codes in Regular Graphs are Difficult. *Discrete Mathematics*, 133:191–205, 1994.

- [20] J. Kratochvíl and M. Krivánek. On the computational complexity of codes in graphs. *Lecture Notes in Computer Science*, 324:396–404, 1988.
- [21] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation, Second Ed.* Prentice-Hall, New Jersey, 1998.
- [22] Chi-Kwong Li and Ingrid Nelson. Perfect codes on the Towers of Hanoi graph. *Bull. Austral. Math. Soc.*, 57:367–376, 1998.
- [23] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes.* North-Holland, Amsterdam, 1977.
- [24] Ingrid Nelson. Coding Theory on the Towers of Hanoi. 1995.
- [25] S.C. Ntafos and S.L. Hakimi. On the complexity of some coding problems. *IEEE Trans. Inf. Theory*, IT-27(6):794–796, Nov 1981.
- [26] Kirk Pruhs. The SPIN-OUT puzzle. *ACM SIGCSE Bulletin*, 25:36–38, 1993.
- [27] Carla Savage. A Survey of Combinatorial Gray Codes. *SIAM Review*, 39(4):605–629, 1997.
- [28] A. Tietavainen and A. Perko. There are no unknown perfect binary codes. *Ann. Univ. Turku*, 148:3–10, 1971.
- [29] Elizabeth Weaver. Gray codes and puzzles on iterated complete graphs. 2005.

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: 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/.

BUCKNELL UNIVERSITY

E-mail address: elizabeth.skubak@bucknell.edu

OREGON STATE UNIVERSITY

E-mail address: stevnich@onid.orst.edu