A NEW PUZZLE BASED ON THE SF LABELLING OF ITERATED COMPLETE GRAPHS

KATHLEEN KING

ADVISOR: PAUL CULL
OREGON STATE UNIVERSITY

ABSTRACT. The Towers of Hanoi puzzle inspired the creation of a perfect one error correcting code on an iterated complete graph of dimension three. This code in turn became was generalized into the SF code, which is a perfect one error correcting code on iterated complete graphs of odd dimension. In this paper, we use the SF code to create a puzzle similar to the Towers of Hanoi. We then proceed to give efficient algorithms to determine the minimum number of moves in which the puzzle can be solved.

1. INTRODUCTION

The Towers of Hanoi puzzle is supposedly based on of a legend. The story goes that Buddhist monks were required to move a stack of 64 sacred disks from one location in their temple to another. There was only one spot in the temple, other than the initial and final places, that was holy enough to hold the disks. The disks were incredibly fragile, so no larger disk could be set on top of a smaller one, and only one could be carried at a time. According to the legend, when the monks finished moving the entire stack to the new location, the world would end [LHS].

The legend inspired Edouard Lucas to create the Towers of Hanoi puzzle, which he published in 1883. Happily, those who took up the problem of the Towers of Hanoi determined that it would require $2^{64} - 1$ moves to shift the entire tower. Thus, even if the monks were extremely efficient and moved one disk each minute, it would take over 3.5 x 10^{13} years to move the entire stack.

Of course, mathematicians being what they are, nobody was willing to let the problem rest, feeling secure in the knowledge that the world will not be ending any time in the near future. Nor were they satisfied with turning the problem into a children's game, in which disks of varying diameter can be placed on three pegs ("towers"), with the same rules as those that bound the monks. No, various aspects of the Towers of Hanoi puzzle have provided ample material to entertain mathematicians.

One problem of interest is the Towers of Hanoi puzzle with more than three towers. Although a Frame and Stewart independently found a solution in 1941, no one has been able to prove that this solution is true [JF],[BS]. Others have found new applications for the traditional three-peg puzzle. Of particular interest to this paper is the work done by Cull and Nelson, who in 1999 created a perfect one error correcting code based on a graph of the Towers of Hanoi [CN].

The graph used in the Towers of Hanoi graph is an iterated complete graph of degree three. Because of the excellent properties provided by the Towers of Hanoi code it inspired others to

Date: August 13, 2004.

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

search for similar codes on iterated complete graphs of degree greater than three. Of the various attempts to create such a code, the most successful has been the SF code for odd degree iterated complete graphs [SK].

This paper uses the SF code to create a new puzzle, one that is similar to the Towers of Hanoi, but has several additional rules and can be played with any odd number of towers. The new puzzle also considers beginning or ending the game at disk configurations other than simple stacks on the various towers. We explain how to solve the puzzle and present an algorithm to find the minimum number of moves necessary to do so.

2. Definitions

Before we proceed, we must present several definitions that will be used throughout this paper.

Definition 2.1. *A **graph**, G, consists of a nonempty finite set V(G) of elements called **vertices** and a finite set E(G) of distinct unordered pairs of distinct elements of V(G) called **edges**. Two vertices, $v_i, v_j \in V(G)$ are **adjacent** if the edge $\{v_i, v_j\} \in E(G)$. A **subgraph** S of G consists of a V(S) $\subset$ V(G) together with the edges connecting any adjacent vertices, $v_i, v_j \in V(S)$.*

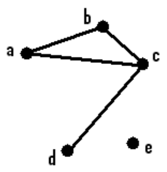

Figure 1. A simple graph

Example 2.2. *In Figure 1, we see a graph G with the vertex set V(G) = $\{a,b,c,d,e\}$ and the edge set E(G) = $\{(a,b),(a,c),(b,c),(c,d)\}$. Any combination of vertices and their associated edges could be a subgraph. One subgraph S has the vertex set V(S) = $\{c,d,e\}$ and the edge set E(G) = $\{(c,d)\}$.*

Definition 2.3. *Two disjoint subgraphs S_1 and S_2 of a graph G are **adjacent** if there exist adjacent vertices in G v_1 and v_2 such that $v_1 \in V(S_1)$ and $v_2 \in V(S_2)$.*

Definition 2.4. *If a vertex v is adjacent to j other vertices, then v has **degree** j.*

Definition 2.5. *A **complete graph on d vertices**, denoted K_d, is a graph which has d vertices and $v_i, v_j \in E(K_d)$ for all $v_i, v_j \in V(K_d)$ with $i \neq j$.*

Example 2.6. *Figure 2 shows the complete graphs on 3, 5, and 13 vertices.*

Definition 2.7. *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 so that exactly one vertex in each K_d^{n-1} subgraph has degree $d-1$ and all other vertices have degree d.*

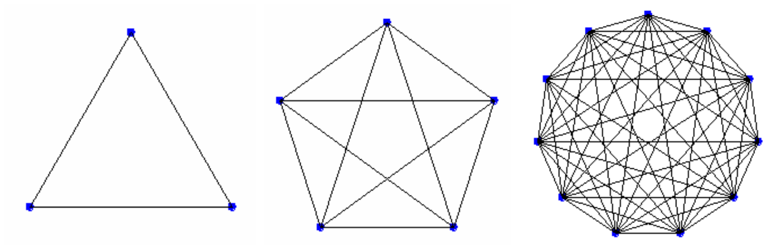

FIGURE 2. The complete graphs K_3, K_5, K_{13}

Example 2.8. *Figure 3 shows the iterated complete graphs K_3^4, which has four iterations on three vertices, and K_7^2, which has two iterations on seven vertices. Note that to construct the next iteration, d copies are made of the current iteration, and these copies are then essentially placed at the vertices of a K_d graph.*

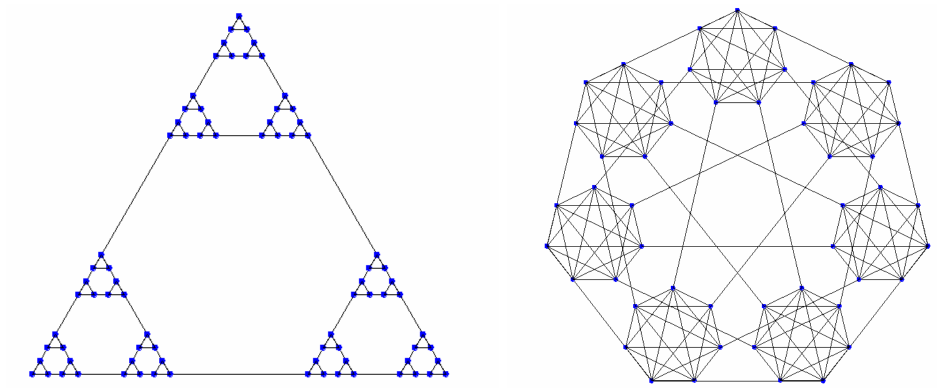

FIGURE 3. The iterated complete graphs K_3^4, and K_7^2

Definition 2.9. *The j^{th}* ***subgraph*** *of a K_d^n graph, with $1 \le j < n$ is a K_d^j graph that is a part of the K_d^n graph. The K_d^j subgraph has* ***order*** *j.*

Definition 2.10. *A* ***corner*** *of a K_d^j subgraph of a K_d^n graph is a vertex that is adjacent to only $d-1$ vertices within the K_d^j subgraph. An* ***external corner*** *of a K_d^j subgraph is a corner that is adjacent to only $d-1$ vertices within the K_d^{j+1} subgraph that contains it. A* ***complete corner*** *is a corner of the K_d^n graph itself.*

Example 2.11. *In Figure 4, the three types of corners are indicated by circles and labelled.*

Definition 2.12. *A* ***code*** *on a graph G is any subset of vertices $C(G) \subset V(G)$. A vertex $c \in C(G)$ is called* ***codevertex****. A vertex $v \in V(G) - C(G)$ is called a* ***non-codevertex****.*

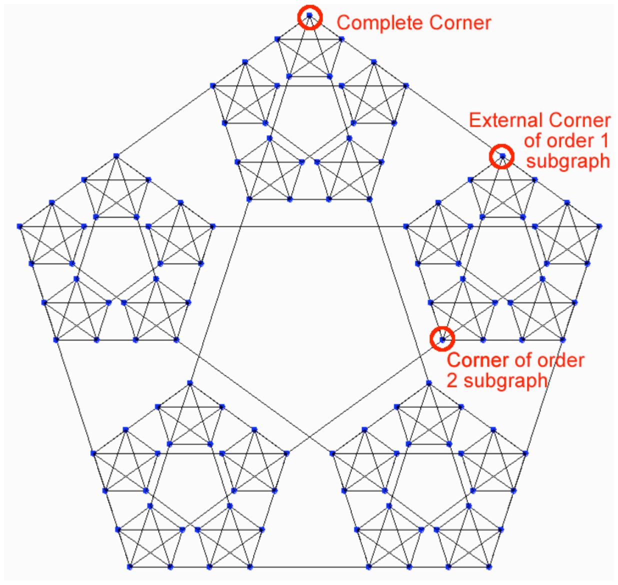
Complete Corner
External Corner of order 1 subgraph
Corner of order 2 subgraph

FIGURE 4. The iterated complete graph K_5^3 with corner types shown

Definition 2.13. *A **perfect one error correcting code** is a code that satisfies the following properties:*

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

3. THE TOWERS OF HANOI CODE

In their paper *Perfect Codes, NP-Completeness, and Towers of Hanoi Graphs*, Cull and Nelson present a family of graphs, such that each graph has a perfect one error correcting code. The graph was inspired by the Towers of Hanoi puzzle; each vertex label represents a configuration of the Towers of Hanoi game, and edges indicate possible moves. Because of this, the vertex labels of each graph naturally correspond to location of the vertex [CN]. Figure 3 shows a Towers of Hanoi graph with vertex strings of length three. Notice that it is the iterated complete graph K_3^3. All Towers of Hanoi graphs are iterated complete graphs of the form K_3^n. The family of graphs is infinite since a graph can be constructed for any value $n \geq 0$.

3.1. **Labelling a Towers of Hanoi Graph.** We will take a moment to examine how the vertex labels correspond to the Towers of Hanoi puzzle. The labels are strings of ternary digits, $\{0,1,2\}$. In a vertex label of length n, $a_1 a_2 \ldots a_n$, each digit, a_i corresponds to a disk in the Towers of Hanoi puzzle. The disks increase in size from the first (leftmost) digit to the last digit. The value of the digit indicates the tower on which the disk is located. So, for example, the label 00120 corresponds to the configuration in which the first, second, and fifth disks are on tower 0, the third disk is on tower 1, and the fourth disk is on tower 2, as shown in Figure 6.

By considering the rules of the Towers of Hanoi puzzle, we know that from position 00120 we can move the first (blue) disk to either of the other towers, or we can move the third (yellow) disk

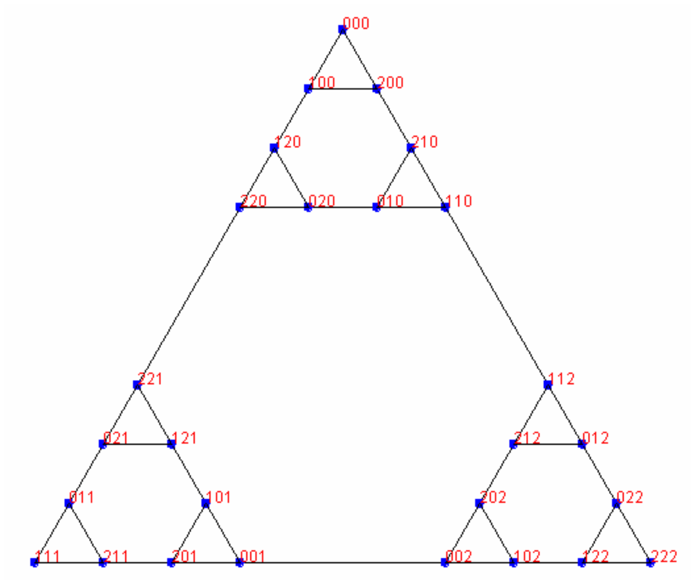

FIGURE 5. The Towers of Hanoi graph for label strings of length 3

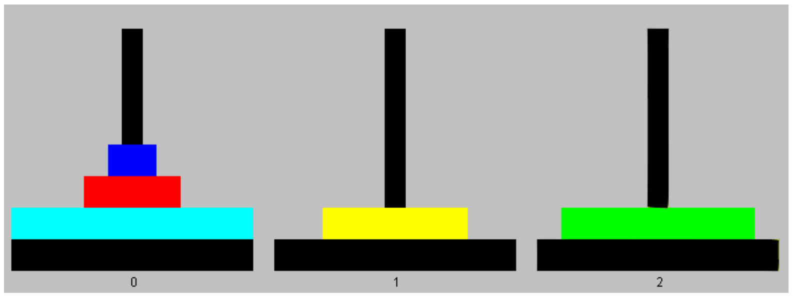

FIGURE 6. The Towers of Hanoi configuration corresponding to label 00120

from tower 1 to tower 2. Thus, the vertex labelled 00120 on the iterated complete graph labelled for the Towers of Hanoi code is adjacent to vertices labelled 10120, 20120, and 00220. These configurations are shown in Figure 7. The graph shown in Figure 8.

For Cull and Nelson, this labelling system is the basis for their code, which they show to have exceedingly desirable characteristics. In particular, they show that a finite state machine for error connection exists, and its size is independent of the length of the label string. They also present a simple method of encoding and decoding. These features make the code an ideal one, but this paper shall not focus on the Towers of Hanoi as a code. We only pause to note its excellent traits because these explain why others have worked to generalize the code.

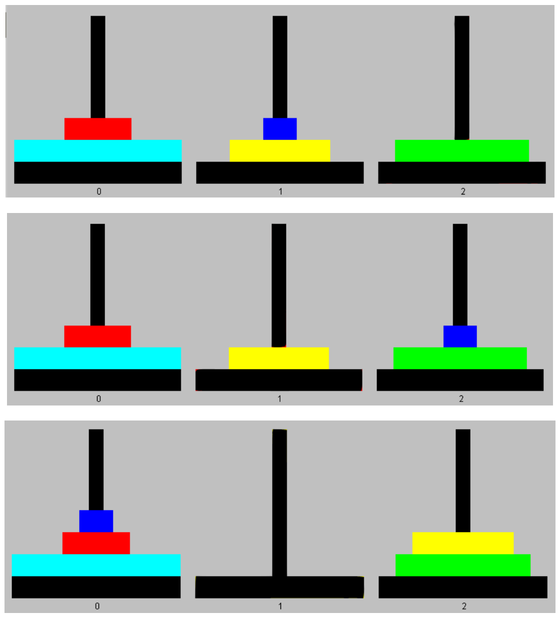

FIGURE 7. The Towers of Hanoi configurations corresponding to labels 10120, 20120, and 00220, respectively

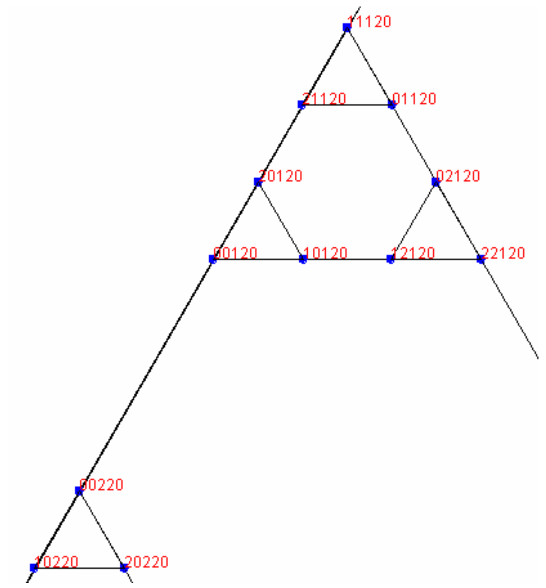

FIGURE 8. Part of the Towers of Hanoi code on K_3^5

4. THE SF CODE

In 2003, Stephanie Kleven created the SF code in an attempt to generalize the Towers of Hanoi code for all odd dimension iterated complete graphs. The work done by Danielle Arett in 1999 indicated that to create a complete Towers of Hanoi code for higher dimensions (i.e., more towers) than three was impractical [DA]. Kleven's code, however, shares several of the desirable properties of the Towers of Hanoi code, including simple finite state machines for codeword detection and error correction and a natural correspondence between vertex labels and locations. For dimension 3, the SF code is the Towers of Hanoi code [SK]. In 2004 Pamela Russell "proved" that the SF code is a subset of the true Towers of Hanoi code for odd numbers of towers; it includes all of the vertices that would appear in a true Towers of Hanoi graph, but only some of the edges. Moreover, Russell showed that it is not possible to create such a code for even dimension iterated complete graphs [PR]. Thus, it appears that the SF code is an excellent generalization of the Towers of Hanoi code, which made it likely that a simple puzzle corresponding to the code could be created.

4.1. **Construction of the SF labelling.** To construct the SF code we begin by labelling K_d^1. The top vertex is 0, and the numbers increase by 1 counterclockwise around the polygon, up to $d-1$. To label K_d^n, each digit, x, of each label in K_d^{n-1} must be permuted by α, where $\alpha(x) = a\left(\frac{d+1}{2}\right)$ mod d. Then, we make d copies of the permuted K_d^{n-1} graph. K_d^n is constructed by rotating the k^{th} copy of K_d^{n-1} by $\frac{2\pi k}{d}$ radians and adding the digit k to the end of each label.

Example 4.1. *An example of the SF labelling construction is shown in Figures 9, 10, and 11.*

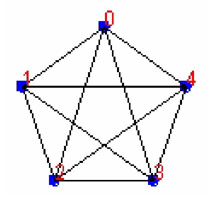

FIGURE 9. SF labelling on K_5^1

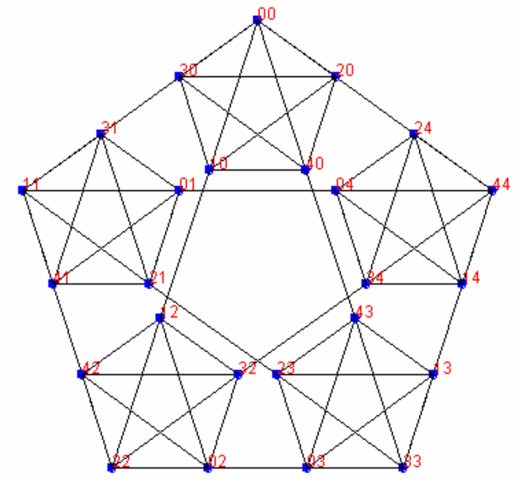

FIGURE 10. SF labelling on K_5^2

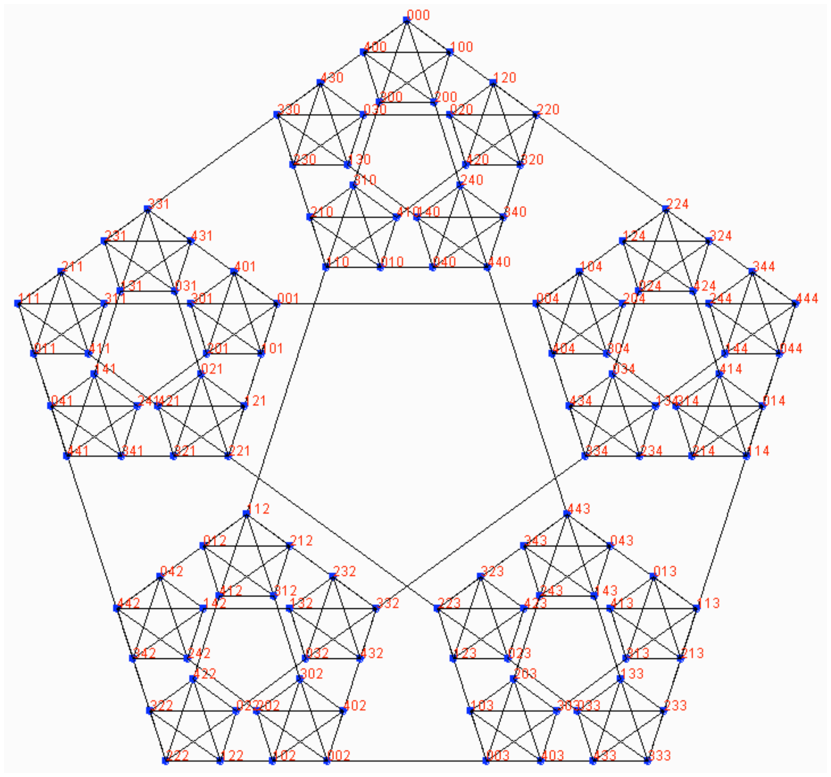

FIGURE 11. SF labelling on K_5^3

4.2. **Observations on SF-Labelled Graphs.** We now proceed to prove some special features of SF-Labelled iterated complete graphs. These will be useful to us in constructing the SF Puzzle. Before we begin, please note that when the "first" digit of a label is mentioned, we mean the leftmost character of the label. Similarly, "last" refers to the rightmost digit. Also, it will be very important to recall from the construction of iterated complete graphs that each K_d^n graph is composed of d K_d^{n-1} graphs, each of which is connected to every other K_d^{n-1} exactly once.

Lemma 4.2. *Within a K_d^j subgraph of a K_d^n graph, all vertices must share the same last $n-j$ digits.*

Proof. We shall prove this lemma using induction. Consider the order n subgraph, K_d^n, that is the graph itself. In constructing it, d copies of K_d^{n-1} were made, with a different digit appended to the end of the label of each vertex, depending on the K_d^{n-1} subgraph to which it belonged. Therefore, all of the labels of the vertices of K_d^n share no ending digits in common.

Now, let us assume that the last $n-j$ digits are the same in any K_d^j subgraph of K_d^n. Consider, then, a K_d^{j-1} subgraph. The last $n-j$ digits of its vertices' labels must be identical, since it is part of a K_d^j subgraph. However, in constructing the K_d^j, a digit is appended to the labels of each K_d^{j-1} subgraph, indicating its position. Since the digit is not the same for all d subgraphs, it is not counted as part of the $n-j$ identical digits shared by all vertex labels in the K_d^j graph. Therefore, the labels of the vertices in a K_d^{j-1} subgraph share the same $n-j+1=n-(j-1)$ last digits.

Thus, since the lemma is true for K_d^n, and its being true for K_d^j implies its truth for K_d^{j-1}, it is true for all cases that the last $n-j$ digits of a K_d^j subgraph of K_d^n must be identical. □

Lemma 4.3. *The labels of the complete corner vertices of a K_d^n graph consist of strings of n identical digits, where the digit indicates the position of the corner.*

Proof. We will use induction to show that this is true. For K_d^1, each label is only one digit long, and the SF Labelling states that one labels the corners of the graph with the numbers from 0 to $d-1$ starting at the top and travelling counterclockwise around the graph. Thus, the lemma is true for $n=1$.

Now, we will assume that it remains true for $n=j$, and show that this assumption implies that it must be true for $n=j+1$. To create the K_d^{j+1} graph, we first make d copies of the K_d^{j+1} graph. Each digit of each label is then multiplied by $\frac{d+1}{2}$. Then, each k^th subgraph is rotated by $\left(\frac{360}{d}\right)k$, which is equivalent to adding $\frac{d+1}{2}$ to each digit of each label in the subgraph. Finally, the digit k is appended to each of label of the k^{th} subgraph. In order for the lemma to be true, the k^{th} corner of the k^{th} subgraph must be $kk...k$ ($j+1$ times). Since the k^{th} corner of the K_d^j graph was labelled $kk...k$ (j times). Then, we can trace the manipulation of this label. Each digit, k, was operated on as follows: $k\left(\frac{d+1}{2}\right)+k\left(\frac{d+1}{2}\right)=2k\left(\frac{d+1}{2}\right)=k(d+1)=kd+k\equiv k \pmod d$. This leaves the k^{th} corner of each K_d^j subgraph with the label $kk...k$ (j times). Then, the digit k is appended to each label in the k^{th} K_d^j subgraph. Thus, the k^{th} corner in the K_d^{j+1} graph has the label $kk...k$ ($j+1$ times), so the lemma is true for $n=j+1$.

Therefore, since the lemma is true for $n=1$ and since its being true for $n=j$ implies its being true for $n=j+1$, the lemma is true by induction. □

Lemma 4.4. *The first j digits in the label of a corner of a K_d^j subgraph of a K_d^n graph must be identical.*

Proof. From Lemma 4.2, all of the vertices in a K_d^j subgraph of a K_d^n graph must share the same last $n-j$ digits. If we consider the construction of the K_d^n graph chronologically, we notice that the first of these identical digits was appended when the K_d^j graph first became a subgraph. Before that, K_d^j was itself simply an SF-labelled iterated complete graph, so it had complete corner vertices whose labels were strings of j identical digits. Then, when K_d^j was incorporated into larger iterated complete graphs, these corner labels were permuted by α, but since $\alpha(k)=\alpha(k)$, for all digits k, the labels of the corners of K_d^j kept their opening strings of j identical digits. Thus, the first j digits in the label of a corner of a K_d^j subgraph must be identical. □

Theorem 4.5. *Each vertex in the SF labelling on a K_d^n graph is adjacent to $d-1$ vertices whose labels have the same last $n-1$ digits as itself but whose first digits are all distinct. Internal vertices are also adjacent to an additional vertex, as defined below:*

Given the label with digits $a_1a_2...a_n$, let $a_1 = a_2 = ... = a_j$, where $1 \leq j < n$. The vertex associated with this label is adjacent to another vertex, whose label is identical except for digit a_{j+1}, which has a value of $2a_1 - a_{j+1}$ (mod d).

Proof. From Lemma 4.2, we know that vertices in the same K_d^1 subgraph share the same last $n-1$ digits. Since the K_d^1 subgraph is simply the complete graph on d vertices, any vertex within it is adjacent to each of the $d-1$ other vertices, all of which differen only in their first digits. Since we see by the construction of a K_d^n graph that all vertices are part of some K_d^1 subgraph, the first part of this theorem must be true.

Now, we consider the second part of the theorem. In iterated complete graphs, only the corners of a K_d^j subgraphs connect to vertices outside that K_d^j subgraph. Lemma 4.4 tells that a vertex's label indicates that it is a corner of a K_d^j subgraph if the first j digits of its label are identical. The vertex that is adjacent must share these first j digits, since the SF Labelling is a Gray code, so adjacent vertices' labels may only differ by one digit. Also, in order to be connected in an iterated complete graph, two vertices that are corners of K_d^j subgraphs must lie within the same K_d^{j+1}, so, from Lemma 4.2, the last $n-j+1$ digits of the vertices must be the same. This leaves only the $j+1^{th}$ digit to differ between the adjacent vertices.

Now, since these adjacent vertices must have the same last $n-j-1$ digits, we will ignore these and consider only labels of the form $aa...ab$, where the first $j-1$ digits are the same, a and the last digit, b, is different. Recall that in the SF labelling, the last digit of a label indicates the position of its subgraph, and the position of a subgraph tells the number of rotations that it has undergone. "Rotating" a subgraph by $\frac{360}{d}$ is equivalent to adding $\frac{d+1}{2}$ to each digit of each label in the subgraph. Therefore, the transformation of a digit z on the K_d^{n-1} graph into a digit on the i^{th} subgraph of the K_d^n graph is $\left(\frac{d+1}{2}\right)z + \left(\frac{d+1}{2}\right)i$.

Recall also that in the SF labelling of the K_d^n graph labels in the j^{th} subgraph are only connected to labels in the j^{th} position of the other $n-1$ subgraphs. So, given a label of the form $aa...ab$, there must be an adjacent label of the form $aa...ac$, where $b \neq c$, such that $c\left(\frac{d+1}{2}\right) + b\left(\frac{d+1}{2}\right) \equiv a$ (mod d). We can solve to find that $c \equiv a\left(\frac{d+1}{2}\right)^{-1} - b$ (mod d). We know that $\left(\frac{d+1}{2}\right)(2) = d+1 \equiv 1$ (mod d), so $\left(\frac{d+1}{2}\right)^{-1} \equiv 2$ and $c \equiv 2a - b$ (mod d).

Thus, we see that a vertex with a label of $a_1a_2...a_n$, with $a_1 = a_2 = ... = a_j$, with $1 \leq j < n$, must be adjacent to another vertex whose label is $a_1a_2...a_j[2a_1 - a_{j+1}(mod\ d)]a_{j+2}...a_n$. □

Example 4.6. *In K_5^4, the vertex 3341 will be adjacent to 0341, 1341, 2341, and 4341, since those vary only in the first digit. Also, since the digits of 3341 are not all the same, it cannot be a corner vertex, and so it will also be adjacent to 33[$3 \cdot 2 - 4(mod 5)$]1 = 3321.*

In K_7^3, the vertex 625 will be adjacent to 620, 621, 622, 623, 624, 626, and 6[$6 \cdot 2 - 2(mod 7)$]5 = 635.

5. The SF Puzzle

Now we are prepared describe the SF Puzzle. It has similar rules to the traditional Towers of Hanoi. One may play with any number of disks and no larger disk may ever be placed on top of a larger one. However, the SF Puzzle is rather more complicated. First, only odd numbers of towers may be used in play. Also, no disk may be moved at all unless all of the disks smaller than it are stacked together, even if other towers are open. Then, even when all of the smaller disks are stacked together, the location of the stack and the current location of the larger disk itself determine where the larger disk may be moved to. In particular, if we let the d towers be numbered 0 through $d-1$ with the stack of smaller disks on tower a and the larger disk on tower b, then the larger disk may only move to tower number $2a-b \pmod{d}$. This may sound rather familiar; it is simply the adjacency rule for labels, Theorem 4.5, transferred back to disks and towers. As with the Towers of Hanoi puzzle, the location of a digit indicates tower number, and the value of a digit indicates disk number.

Example 5.1. *The label 4400 on a K_5^6 graph corresponds to the configuration shown in Figure 12 below. Tower number 3 is colored yellow because the the yellow disk may be placed there next, since $2(4)-0=8\equiv 3$ (mod 5). Of course, the smallest disk may always move to any tower, since there are no smaller disks to stack and regulate its movement. Note that this ability to move the smallest disk anywhere corresponds to the adjacency of a label to all of the labels with different first digits, as described in Theorem 4.5.*

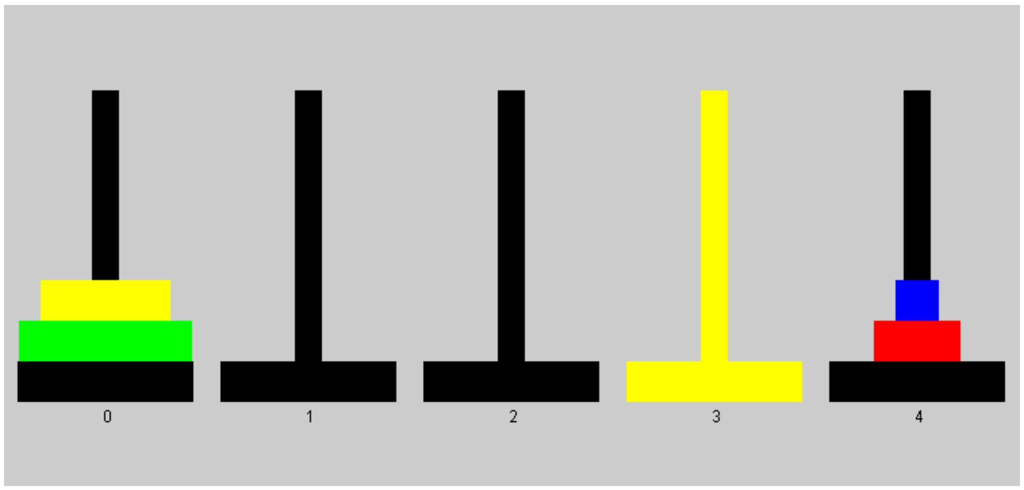

FIGURE 12. The SF Puzzle configuration for the label 4400

One interesting feature of the SF puzzle is that it is sometimes more challenging to play when the starting and ending configurations are not simply stacks of all of the disks. Given any two configurations, one can attempt to move from one to the other using the minimum number of moves.

6. Solving the SF Puzzle

Because the moves and configurations of the SF puzzle map to the edges and vertices of an SF-labelled iterated complete graph, we know that it must always be possible to solve the puzzle, no matter what starting and ending positions are given. To solve the puzzle requires recursive thinking. One must begin by considering to what tower the largest disk must be move. In order to get the largest disk to its end location, where do all of the other disks need to be stacked? And in order to get the second largest disk to this point, where must the other $n-2$ disks be placed? One follows this reasoning until the placement of the top disk is determined, and then one can begin moving all of the others.

A natural question to ask is how many moves it will take to solve the puzzle. We can prove that the minimum number of moves necessary to shift the entire stack of n disks on d towers is $2^n - 1$, which we do in subsection 6.1. However, it is also interesting to consider the case in which the puzzle is played with different starting and ending positions. In order to solve this problem, we create an algorithm, which is given in subsection 6.2.

6.1. **The Minimum Number of Moves to Solve the Traditional SF-Puzzle.** We first consider solving the SF-Puzzle in the traditional Towers of Hanoi style. That is, we move a stack of n disks from one of d towers to another. This problem is equivalent to finding the distance between two vertices on an SF-labelled graph, where the vertices correspond to different single stack configurations of the puzzle. Now, from Lemma 4.3, we know that labels on the corners of K_d^n graphs consist of strings of n identical digits. Recall from Section 3 that when all of the n disks of a puzzle are on tower number j, the corresponding label is a string of n j's. Thus, we may convert the problem of finding the minimum number of moves between stacks to finding the distance between two corner vertices on the K_d^n graph. We will now solve this problem.

Theorem 6.1. $2^n - 1$ *is the shortest length for a path that connects any two distinct complete corner vertices on the* K_d^n *graph. Moreover, this path is unique.*

Proof. We will prove this using mathematical induction.

For $n = 1$, there is clearly a unique shortest path of length $2^1 - 1 = 1$ between any two corner vertices on K_d^1 since K_d^1 is simply the complete graph on d vertices, so one edge connects each vertex to every other vertex and a path of length 1 is the shortest possible between two distinct vertices.

Now, let us assume that there exists a unique shortest path of length $2^{n-1} - 1$ for K_d^{n-1}. Note that in order to get from one complete corner to another we must first travel from the complete corners to external corners of the same K_d^{n-1} subgraphs because complete corners are not adjacent to vertices outside their own subgraphs. Thus, the the shortest path must start from one of the complete corner vertex in one K_d^{n-1} subgraph, pass through an external corner in the same subgraph, and then somehow move to an external corner of the destination K_d^{n-1} subgraph before finally getting to the final corner vertex. From the inductive hypothesis, we know that the lengths of the paths between the complete and external corners of the K_d^{n-1} subgraphs must each be $2^{n-1} - 1$ and, for two given corners, these paths are unique.

Now, we must consider how to travel between the two subgraphs. The shortest distance between any two distinct vertices in a graph is 1. Since iterated complete graphs allow exactly one edge between every pair of subgraphs, we know that a unique path of length 1 must exist to connect the two subgraphs. Also, this edge determines a second corner in each of the two subgraphs that must be hit by the path. Thus, we know two corners in each K_d^{n-1} subgraph, so from our assumption, a unique shortest path can be found in each subgraph, and, when these pieces are put together, the complete shortest path between the corner vertices is unique.

The length of this shortest path is simply the lengths of the paths through the two subgraphs added to one, for the connecting edge. From the inductive hypothesis, the length of shortest path through the subgraph is $2^{n-1}-1$. So, we can now find that the length of the shortest path through the K_d^n graph is $2(2^{n-1}-1)+1 = 2^n-2+1 = 2^n-1$.

So, by mathematical induction, we find that 2^n-1 is the shortest length for a path that connects any two distinct corner vertices on the K_d^n graph and this path is unique. □

6.2. **The Minimum Number of Moves to Solve the General SF Puzzle.** In the previous subsection we showed that it is possible to solve the SF puzzle in 2^n-1 moves, if one is moving all of the disks from one tower to another. However, we mentioned earlier that a more interesting approach to the SF puzzle is to consider starting and ending positions other than simple stacks. Since each position of the SF puzzle corresponds to a vertex on an SF-labelled graph, we can approach this problem by considering the distance between any two vertices on an SF-Labelled iterated complete graph. In this subsection we discuss an efficient algorithm that solves the problem. This algorithm uses Theorems 4.5 and 6.1 to calculate distance. Pseudocode for the algorithm is shown below. There are two parts. The first is all of the code necessary to find the total distance between two vertices whose labels are *labelA* and *labelB*. The second finds the distance from a given *label* to the a^{th} corner of its K_d^N subgraph. We now present this algorithm.

```
Method FindDistance(labelA, labelB)
  bestDistance = $2^n$-1
  start = n-1
  aLast = labelA(start)
  bLast = labelB(start)
  while(aLast == bLast)
    start = start - 1
    aLast = labelA(start)
    bLast = labelB(start)
  count = 0
  while(count < d)
    if(count $\neq$ aLast)
      distance = ToCorner(0, labelA, count, start)
      if(count == (aLast+bLast)/2 mod d)
        distance = distance + 1 + ToCorner(0, labelB, count, start)
      else
        Roundabout = (2(count)+bLast-aLast)/2 mod d
        distance = distance + $2^{n-1}$+ 1 + ToCorner(0, labelB, Roundabout, start)
      if(distance < bestDistance)
        bestDistance = distance
    count = count + 1
```

```
Method ToCorner(previousDistance, label, a, N)
  if(N > 0)
    if(a != label(N-1)
      a = (a + (label(N-1))/2 mod d
      previousDistance = previousDistance + $2^{N-1}$
    ToCorner(previousDistance, label, a, N-1)
```

We must make several comments about the format of the code. “$x/2$” here means $x \cdot 2^{-1}$ (mod d). So, for $d = 5$, $2^{-1} = 3$, and $x/2 = 3x$. Also, please assume that `ToCorner` returns the final distance that it finds. In practice, these methods were written in Java, and a private distance variable was made available to both the `ToCorner` and `FindDistance` methods.

Now, we are free to show that the find distance algorithm does, in fact, find the distance correctly.

Theorem 6.2. *The `ToCorner` algorithm given above correctly finds the shortest distance between a vertex and the a corner of the graph of order N in which it is located.*

Proof. We shall prove the theorem using induction. Let $P(i)$ be the proposition that the `ToCorner` algorithm gives the correct distance from a corner of the K_d^i subgraph to any vertex within it.

We must show that $P(1)$ is true (i.e., the algorithm work correctly for K_d^1). Recall that a K_d^1 subgraph is simply the complete graph on d vertices. Thus, the distance from any vertex of the graph to any corner is either 0 (if the vertex and the corner are coincident) or 1 (since all vertices on the complete graph are connected). We can show that the `ToCorner` algorithm finds the distance correctly in either case.

`ToCorner` is given several initial variables. First is *previousDistance*, which tells the distance already travelled on the path from the original corner to the vertex. Then, *label* simply tells the name of the vertex, *a* declares the corner to which we are calculating distance, and *N* tells the degree of the current subgraph, which is 1 in this instance.

When `ToCorner` runs with $N = 1$, we see that if $a = label(0)$ (the 0^{th}, or first, digit of *label* is the same as *a*), then nothing is done but calling `ToCorner` again, this time with $N = 0$, which ends the program on the run through. Since we know that the corner indicated by *a* and the vertex to which the *label* belongs are in the same K_d^1 subgraph, if they have the same first digit, they are coincident. Therefore, the distance between them is 0, and the total distance should remain the *previousDistance*, as entered into the method. The algorithm does this correctly.

Now, if *a* is not the first digit in the label, the vertex must have a distance of one from the corner. The algorithm sees that the two are not equal, and so it adds $2^{1-1} = 2^0 = 1$ to the *previousDistance* before calling itself again with $N = 0$ to end the program. This also is correct. (Note that although *a* is also affected when the corner and vertex were not coincident, this change has no effect on the running of the program since it ends immediately thereafter.)

Thus, we have found that $P(1)$ is true. We now assume that $P(j-1)$ is true and use this to show that $P(j)$ must also be true.

`ToCorner` will first check to see that $j > 0$. Once assured of this, it compares *a* and $label(j-1)$. If, $a = label(j-1)$ then the vertex to which the *label* belongs is in the same K_d^{j-1} subgraph as the corner. Thus, it requires no additional moves to reach the corner of the *label*’s K_d^{j-1} subgraph, so no distance is added. The algorithm simply calls itself, this time with $j-1$ rather than j, and by the inductive assumption we know that the algorithm must find the correct distance, so $P(j)$ is true.

It is also possible, though, that $a \neq label(j-1)$. Then the destination vertex is not in the corner's K_d^{j-1} subgraph (because we know from Lemma 4.2 that vertices in the same K_d^{j-1} subgraph share the same last $n-(j-1)$ vertices). So, to find the path from the corner to the vertex, we must first move to the vertex's K_d^{j-1} subgraph. To do this, we travel to a different corner of the beginning corner's K_d^{j-1} subgraph, since our original corner must be an external corner because it is also the a corner of a K_d^j subgraph. From theorem 6.1, we know that the distance between two corners of a K_d^{j-1} graph must be $2^{n-1}-1$. Since we want the shortest complete path, we choose a corner that is adjacent to the K_d^{n-1} subgraph containing the destination vertex, so that the total distance from the initial corner to a corner of the destination vertex's K_d^{n-1} subgraph is $(2^{n-1}-1)+1 = 2^{n-1}$. This is the amount of distance added on by the `ToCorner` algorithm.

However, we must know how to choose the corner adjacent to the destination subgraph because only one of the $d-1$ non-external corners is adjacent to the destination vertex's subgraph. Fortunately, Lemma 4.2 tells that corners of K_d^{j-1} subgraphs share the same last $n-j-1$ digits. This means that digits 0, 1, ... , $j-2$ may be different, but digit $j-1$ of every vertex in the subgraph will be identical. Also, from Theorem 4.5 we know the form of the label for adjacent vertices that do not lie within the same K_d^1 subgraph. Now we wish to travel from our initial corner, whose label we shall call $aa...abc...$ (where there are j a's followed by a string of $n-j$ other digits) to the corner in the destination vertex's subgraph with the label whose j^{th} digit is $label(j-1)$ (since we start counting digits at zero).

We first travel from our starting corner $aa...abc...$ to another corner in the same subgraph, which from Lemma 4.4 must have the form $xx...xabc...$ (where there are $j-1$ x's). The corner labelled $xx...xabc...$ is adjacent to the corner in the subgraph of our destination vertex, which we already said is labelled $xx...x[label(n-1)]...$. From Theorem 4.5, we know that the j^{th} digit of these two labels must have a special relationship, so we can write the equation $2x-a = label(j-1)$ and solve to find that $x = (label(j-1)+a)/2 \bmod d$. The corner of the destination vertex's K_d^{j-1} subgraph with the label satisfying these requirements becomes the next corner to be input to the `ToCorner` algorithm, so the algorithm sets its a variable to this new value. Having accomplished this, the `ToCorner` algorithm calls itself with its adjusted *previousDistance*, a, and $j-1$. Then, we know from our inductive assumption that the `ToCorner` algorithm works correctly for the K_d^{j-1} subgraph. Since we have shown that for K_d^j `ToCorner` does, in fact, increment the distance correctly and set the value of the next corner appropriately, we know that the algorithm is prepared to work for K_d^{j-1}. So, $P(j)$ is true.

Thus, since $P(1)$ is true, and since $P(j-1)$ true implies that $P(j)$ is true, we can declare that the `ToCorner` algorithm given above correctly finds the shortest distance between a vertex and the a corner of the K_d^N subgraph in which it is located. □

Example 6.3. *Let us look at how the* `ToCorner` *algorithm works when it is called as* `ToCorner(0,` *0231, 4, 3) inside a K_5^4 graph. This means that we are trying to find the distance from the vertex labelled 0231 to the 4 corner of its K_5^3 subgraph. The 4 corner of the K_5^3 subgraph containing 0131 is 4441. Now, we shall trace through the algorithm. Since $N = 3 > 0$ and*

$previousDistance$	$label(N-1)$	a	N
0	3	4	3
4	2	1	2
6	0	4	1
7	-	2	0

TABLE 1. The values of variables in the `ToCorner` algorithm

$label(3-1) = label(2) = 3 \neq 4 = a$, we adjust the values of a and previousDistance. Thus, $a = (label(2)+a)2^{-1} = (3+4)(3) = 21 \equiv 1$ (mod d) and $previousDistance = 0+2^{3-1} = 4$.

Now `ToCorner` is called again, but this time with a new set of variables: `ToCorner(`4, 0231, 1, 2). Thus, we are now finding the distance from the vertex labelled 0231 to the 1 corner of its K_5^2 subgraph, which is labelled 1131. Again, we see that $2 > 0$ and $label(1) = 2 \neq 1$, so we find new values for a and previousDistance, as shown in Table 1. The algorithm is then called again with the new values for previousDistance, a, and N, and since since $N = 1 > 0$ and $label(0) = 0 \neq a = 4$, a and previousDistance are adjusted once again, to the values in Table 1. The program then calls `ToCorner(`7, 0231, 2, 0), but since $N = 0$, $N \not> 0$ and the program ends with a final distance is 7. We can confirm this by looking at a graph of K_5^4, which is shown in Figure 13.

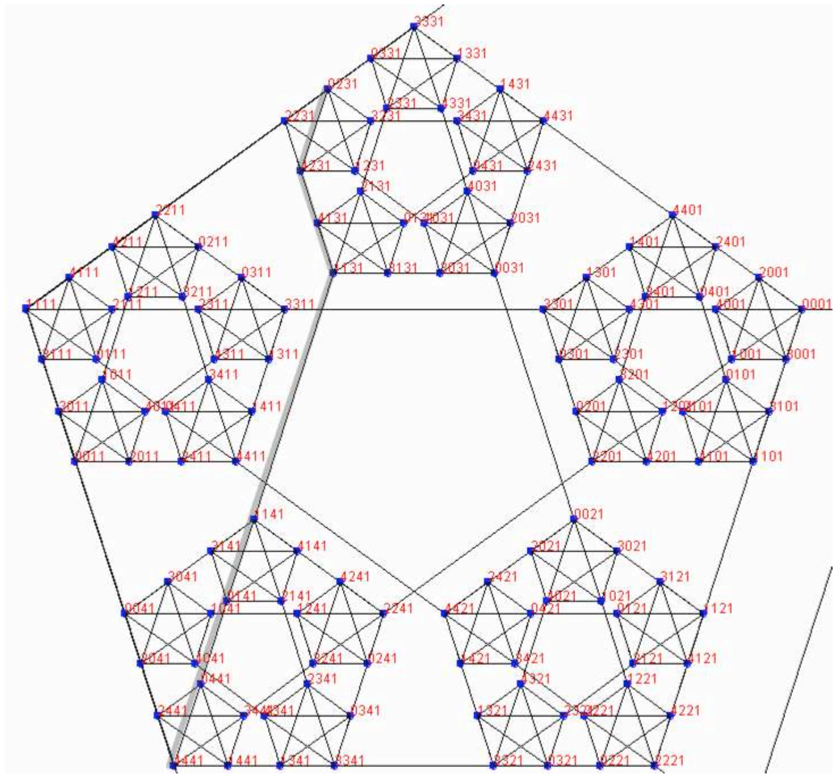

FIGURE 13. Part of the SF Labelled K_5^4 graph, showing the path from 4441 to 0231 measured by the `ToCorner` algorithm

Theorem 6.4. *The* `FindDistance` *algorithm given above correctly finds the shortest distance between two vertices on an SF-labelled K_d^n graph.*

Proof. To travel from one vertex to another on a K_d^n graph, we must first determine whether the two vertices lie within any of the same subgraphs. This is indicated by the label, since Lemma 4.2 states that labels with the same last j digits lie within the the same K_d^{n-j} subgraph. Thus, to find the distance between two points, we determine whether they share any common digits. These j digits are then ignored, and the problem is considered within the K_d^{n-j} graph. The `FindDistance` algorithm performs this task of looking for common subgraphs in the first while statement, which determines how many digits are shared by the two labels. It sets the *start* value accordingly to $n-j-1$, which is the highest order of subgraph which does not contain both vertices. It also sets the values of *aLast* and *bLast* to be the last non-shared digits of the two labels.

Now, we have two vertices, let us call them A and B, each lying in different K_d^{start} subgraphs of the same K_d^{n-j} subgraph, so that the label of A is that which contains *aLast* and similarly B. To travel from A to B, we must move from one K_d^{start} subgraph to the other. Since subgraphs are only connected to other subgraphs by their corners, we must travel from A to an corner of its K_d^{start} subgraph, then somehow cross to an corner of B, and finally proceed from that corner to B itself. Of course, we have many choices here; how do we choose the corners, and how do we connect them?

The `FindDistance` algorithm calculates the distance from A and B to each corner in their subgraphs, except for the subgraphs' external corners, which by definition are not adjacent to any other K_d^{start} subgraph within the same K_d^{n-j} graph. The distance from A to each of its corners is calculated first, using the `ToCorner` algorithm, since we know from Theorem 6.2 that `ToCorner` correctly finds the distance from a vertex to an external corner of its the K_d^{start} subgraph. We then find the closest corner B's subgraph, and calculate the sum of all of the distances (from A to its corner, between the two corners of the subgraphs, and from B to its external corner).

Of course, we have not yet explained how the external corners of A and B are matched, or how the distance between these corners is calculated. Since we know from the construction of iterated complete graphs that there is one corner of subgraph A that is adjacent to a corner of subgraph B, we will consider this easy case first. Since these two corners are adjacent, the distance between them is 1. Then, using Theorem 4.5, we can find which corners these are within their respective K_d^{start} subgraphs. Because these adjacent vertices are corners of K_d^{start} subgraphs, their first *start* digits must identical. Since the SF Labelling is a Grey code, only one digit can vary in the labels of adjacent vertices, so the strings of *start* identical digits at the beginning of each corner's label must be the same as one another. (Because if one digit in the string varied, all of the digits would vary, and unless this is the trivial case, that would make more than one digit different between adjacent labels.) Recall also that since the A and B subgraphs are part of the same K_d^{n-j} graph, Lemma 4.2 states that the last j digits must be the same for vertex A, vertex B and all of the other vertices in the subgraph. This leaves only the number *start* digit of the adjacent external corners to be differ between the two labels. So, the external corner of subgraph A that has a label of the form $xx...x[aLast]...$ is adjacent to the corner of subgraph B with $xx...x[bLast]...$, where x is some digit and *aLast* and *bLast* are the number *start* digits in each label. From Theorem 4.5, we know that $2x - aLast = bLast$. Thus, the distance between the A and B subgraphs is 1 if we are connecting the $x = (aLast + bLast)/2 \bmod d$ corners of each subgraph.

To travel from any other corner of A to any other corner of B, we must "cut across" other subgraphs. To keep these paths as short as possible, we pair corners of subgraphs A and B that are adjacent to the same K_d^{n-j-1} subgraph. Thus, from Theorem 6.1, the distance between the two corners is only $1+(2^{start}-1)+1=2^{start}+1$, since it is a distance of 1 between the external corner of subgraph A and its adjacent corner, a distance $2^{start}-1$ between corners of the same K_d^{start} subgraph, and another distance of 1 between the external corner of subgraph B and its adjacent corner.

Thus, we know the distance between nonadjacent external corners of subgraphs A and B, but we must still determine how to pair these corners. In the `FindDistance` algorithm, we find the distance to A to one of the external corners of its K_d^{start} subgraph, and then pair it to the appropriate corner in subgraph B. If the external corner of A's subgraph is not directly adjacent to any corner of B, we know that we must cut across another subgraph. So, we must find the external corner of subgraph B that is adjacent to the same subgraph as the *count* corner of subgraph A. For ease of writing, we will let $k = count$. Now, as described earlier, adjacent corners of K_d^{start} subgraphs have the differ only in the *start* digits of their labels. Thus, from Theorem 4.5, the *count*, or k, corner of subgraph A is adjacent to a corner in another subgraph with a label of the form $kk...k[2k-aLast]...$, where $2k-aLast \pmod d$ is the value of the *start* digit. Similarly, there exists some corner of subgraph B that is adjacent to a corner in the same subgraph with the a label of the form $xx...x[2x-bLast]...$, where $2x-bLast \pmod d$ is the value of the *start* digit. However, since both the corner labelled $kk...k[2k-aLast]...$ and the corner labelled $xx...x[2x-bLast]...$ are in the same K_d^{start} subgraph, we know from Lemma 4.2 that $2x-bLast = 2k-aLast$. We solve to find that $x=(2k-aLast+bLast)/2 \bmod d$. This means that the x corner of subgraph B is adjacent to the same subgraph as the *count* corner of subgraph A, and we have already determined that the distance between these corners of subgraphs A and B is $2^{start}+1$.

Within its second while loop, the `FindDistance` algorithm calculates the distance from A to its subgraph's corners using `ToCorner`, matches these corners with the appropriate corners of subgraph B, and finds the distance from B to the correct corners. It adds up the total *distance* of each path, and compares each of these values to the *bestDistance*, replacing the *bestDistance* with *distance* if the *distance* is shorter. Thus, in the end, the *bestDistance* is indeed the shortest distance, so we see that the `FindDistance` algorithm does correctly calculates the distance between two given vertices on an SF-labelled K_d^n graph. □

Example 6.5. *We will look at the workings of the* `FindDistance` *algorithm on the* K_5^3 *graph, which is shown in Figure 11. We will let* $labelA = 424$ *and* $labelB = 331$*. The code initially sets* $bestDistance = 2^3-1 = 8-1 = 7$ *and* $start = n-1 = 3-1 = 2$*, so* $aLast = labelA(2) = 4$ *and* $bLast = labelB(2) = 1$*. Since* $4 \neq 1$*, we do not change the values of* $aLast$*,* $bLast$*, or* $start$*. Now we set count to 0 and enter the second while loop.*

Since $count = 0 \neq 4 = aLast$*, we set distance to* `ToCorner`$(0,424,0,2)$*, which returns 2, which we can confirm by looking at the graph* K_5^3 *shown in Figure 11 and checking the distance between vertex 424 and vertex 004. Then, we check to see whether count is* $(aLast+bLast)\left(\frac{d+1}{2}\right)$*. In this case,* $(4+1)\left(\frac{5+1}{2}\right) = 15 \equiv 0$ *(mod d), which is equal to* $count$*. This means that the corner 004 is adjacent to a corner of labelB's subgraph, namely 001. The vertex labelled with labelB is a distance of* `ToCorner`$(0,331,0,2) = 3$ *away from 001, so the total distance of this path is*

$3+1+3=7$. *Since bestDistance was already set to 7, the calculated path is no shorter, and bestDistance is not changed. The count variable is then incremented and the process repeated.*

This time, count $=1$, *which is still not equal to 4, so we find that the distance from 424 to 114 is 2, and we set distance accordingly. Since 114 is not directly adjacent to the subgraph containing labelB, we must find out which corner of labelB's subgraph is adjacent to the same subgraph which is adjacent to 114. We find that* $Roundabout = (2(count)+blast-aLast)\left(\frac{d+1}{2}\right) = (2(1)+1-4)\left(\frac{5+1}{2}\right) = -3 \equiv 2$ *(mod d). So, we now must find the distance between the vertices labelled 221 and* 331 *using the* `ToCorner` *algorithm, which we can see by inspection must be* $2^2-1=3$, *since 221 and 331 are corners of an order 2 subgraph (Theorem 6.1). This gives a total distance of* $2+(2^2+1)+3=10$, *which is more than the current bestDistance of 7. We continue this process, until we have considered the paths through of the 4 corners of labelA's subgraph which are connected to other subgraphs. As it turns out, the path through 224 is the shortest, with a length of 6.*

We now know that the `FindDistance` and `ToCorner` algorithms correctly determine the minimum number of moves necessary to solve the SF puzzle. However, many algorithms exist to solve every (solvable) problem. We wish to find an algorithm that not only works, but is also efficient. Thus, we must also consider the run time of each algorithm. We desire that the time needed to run the algorithm grows linearly with respect to n and with respect to d.

Lemma 6.6. *The* `ToCorner` *algorithm given above has a run time of* $\Theta(N)$.

Proof. The `ToCorner` algorithm performs a constant number of calculations each time it calls itself, and it calls itself N times. Therefore, `ToCorner` has a run time of $\Theta(N)$. □

Theorem 6.7. *The* `FindDistance` *algorithm given above has a run time of* $\Theta(dn)$.

Proof. The `FindDistance` algorithm begins with several lines that are run only once, each of which take constant running time. Now, in the worst case for running time, the vertices are in different K_d^{n-1} subgraphs of the K_d^n graph. In this case, the last digits of the two vertex labels are different, so the while loop conditional is also checked only once.

We now enter a for loop that runs d times. Within in this loop are several steps that use only constant time for each pass through of the loop. However, `ToCorner` method is called twice during each pass. Lemma 6.6 states that the `ToCorner` method itself has a run time of $\Theta(N)$. Since in this case, the initial value for N is *start*, and, for the worst case, we allow $start = n-1$. So, the run time of `ToCorner` is $\Theta(n-1)=\Theta(n)$.

So, the `FindDistance` algorithm has a number of constant calculations, a number of d times constant calculations, and $2d\Theta(n)=\Theta(2dn)=\Theta(dn)$ calculations. Since we only look at the leading term to determine run time, we can see that the `FindDistance` algorithm has a run time of $\Theta(dn)$. □

7. Discussion

We have presented a study of the SF labelling, used this study to create a new puzzle, found the minimum number of moves needed to solve the standard puzzle, and presented an algorithm to find the minimum moves for the puzzle with any starting and ending configurations. We also showed that the algorithms used to calculate the minimum number of moves has an efficient run time.

We did not give an algorithm to find the sequence of moves that solves the SF puzzle. However, this could be done easily by using Theorem 4.5 and building off of the current algorithms, which already determine the corners of the subgraphs through which the path travels. To solve the puzzle one would only need to write a simple program to determine the paths between the various corners.

We also do not address the amount of computer memory used by each algorithm, although we believe that the algorithms presented use an acceptable amount of space. However, since we have not given any proof of the minimum possible run time or space we cannot say with certainty that the distance finding algorithm has maximal efficiency.

References

[DA] Arett, Danielle. *Coding Theory on the Generalized Towers of Hanoi.* Proceedings of the REU Program in Mathematics. NSF and Oregon State University. Corvallis, Oregon. August, 1999.

[CN] Cull, Paul and Ingrid Nelson. *Perfect Codes, NP-Completeness, and Towers of Hanoi Graphs.* Bulletin of the Institute of Combinatorics and its Applications, Volume 26, 13-38. 1999.

[JF] J. S. Frame. *Solution to advanced problem 3918.* American Mathematics Monthly, 48 (1941), pp. 216217.

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

[LHS] Lawrence Hall of Science. *Towers of Hanoi.* University of California, Berkeley. Website url http://www.lhs.berkeley.edu/Java/Tower/towerhistory.html. July 29, 2004.

[PR] Russell, Pamela. *Perfect One Error Correcting Codes on Iterated Complete Graphs: Encoding and Decoding.* Proceedings of the REU Program in Mathematics. NSF and Oregon State University. Corvallis, Oregon. August, 2004.

[BS] B. M. Stewart. *Solution to advanced problem 3918.* American Mathematics Monthly, 48 (1941), pp. 217219.

FRANKLIN W. OLIN COLLEGE OF ENGINEERING
E-mail address: `kathleen.king@students.olin.edu`