Alternate Labelings for Graphs Representing Perfect-One-Error-Correcting Codes

David Bode [1]
St. Olaf College
1500 St. Olaf Avenue
Northfield, MN 55057

Advisor: Professor Paul Cull
Department of Computer Science
Oregon State University
Corvallis, Oregon USA
97331-3202
pc@cs.orst.edu

August 24, 2003

[1]The exceptional guidance of Professor Paul Cull in developing and refining the ideas contained within this paper is gratefully acknowledged.

Abstract

Alternate labelings for one- and two-dimensional graphs which support perfect-one-error-correcting codes are provided, and their "efficiency" is compared with previously-known labelings using the concept of the finite state machine. These labelings are produced by altering the finite state machines used for recognition and error-correction for the previously known labelings. It is discovered that the Standard Binary labeling in one dimension and the Towers of Hanoi labeling in two dimensions are not unique finite state labelings, but they remain the most efficient labelings in their respective dimensions.

Chapter 1

1.1 Introduction

No one can deny the high degree of technology which the computers and communications networks of today have acquired. Yet due to human construction, faulty hardware, or other adverse conditions, errors may still occur in computer transmissions. Given the importance of computers and the information transmitted by them in our society today, the ability to detect and correct these errors is essential. One way of insuring error-detection and correction involves the use of error-correcting codes, which can be studied as labels on a graph consisting of vertices and edges. Previous work in this area has revealed that there are at least two perfect one-error-correcting codes on the one-dimensional graph, namely the Standard Binary labeling and the Gray labeling, and that there is at least one perfect one-error-correcting code on the two-dimensional graph, namely the Towers of Hanoi labeling. It has been conjectured that the Standard Binary labeling in one dimension and the Towers of Hanoi labeling in two dimensions are unique labelings, in their respective dimensions, which are finite state for all three tasks of recognition, error-correction, and decoding. We examine other possible means of labeling the one-dimensional and two-dimensional graphs by looking at methods of altering the finite state machines which carry out the tasks of recognition and error-correction for the known labelings. We reveal alternate labelings which are finite state for all three tasks, derived using these methods, thus refuting the conjecture that the Standard Binary and Towers of Hanoi labelings are unique finite state labelings.

1.2 Background and Definitions

1.2.1 Error-Correcting Codes

An error-correcting code on a graph consists of a subset of the vertices (these special vertices are called codewords) and a rule that given any vertex returns the codeword vertex nearest to the given vertex. Distance here is defined in the obvious way as the number of graph edges which are traversed on the shortest path between the two vertices. For this rule to make sense, there should be only one codeword which is closest to each vertex. When for some vertices there are two or more closest codewords, the code is said to detect an error, but it can't correct the error.

If each codeword is at distance 2d+1 from another codeword, then we can think of a codeword as sitting at the center of a 'sphere' of radius d. Each vertex in this sphere will be decoded as the codeword. When every vertex is in exactly one of these spheres, the code is a perfect d-error- correcting code. In particular, in a one-error-correcting code each codeword should be at distance 3 from another codeword. If the spheres of radius 1 around each codeword do not overlap and do include every vertex, then the code is a perfect one-error-correcting code. [1] In other words, every vertex is either a codeword, or shares an edge with a codeword. For our purposes, we would like the or to be an exclusive one.

1.2.2 The Finite State Machine

The concept of the finite state machine can be represented by a collection of appropriately labelled points, which correspond to various states of being, and arrows, which connect the points, indicating movement to another state of being upon varying inputs and possibly producing outputs with each move. To use a finite state machine, one begins at the state marked "start" and reads an input string digit by digit. After reading the first digit, one follows the arrow corresponding to the digit read to another state (or possibly the same state). At the new state, one reads the next digit, following the corresponding arrow again. This process is repeated until all of the digits in the string have been read. At this point, a characteristic of the string read may be determined by observing which of the states one is in. A finite state machine with n states is called an n-state machine.

From here on, we will call any task "finite state" if it can be accomplished by a finite state machine. A finite state task's "simplicity" may be measured by looking at the value of n for the n-state machine which accomplishes that task. Smaller values of n will correspond to greater simplicity and larger values of n will correspond to greater difficulty. We will call a labeling "finite state" if all three of the tasks of recognition, error-correction, and decoding are finite state for that labeling. The "efficiency" of a finite state labeling will be measured by the sum of the values of n for each of the finite state machines used to accomplish the three tasks for that labeling (i.e. the sum of the simplicity values for that labeling).

1.2.3 Recognition

Recognition is the term used to describe the process of determining the type of word to which a string corresponds. The finite state machine used to carry out the process of recognition is called a recognizer. Thus, a recognizer will indicate if a given string is a codestring or a non-codestring and, given a non-codestring, may indicate which type of non-codestring it is.

1.2.4 Error-Correction

Error-correction is the process of taking a non-codeword and sending it to the nearest codeword. This process involves altering the corresponding non-codestring in such a way as to change it to the nearest codestring. The finite state machine which carries out this process is called an error-corrector.

1.2.5 Encoding and Decoding

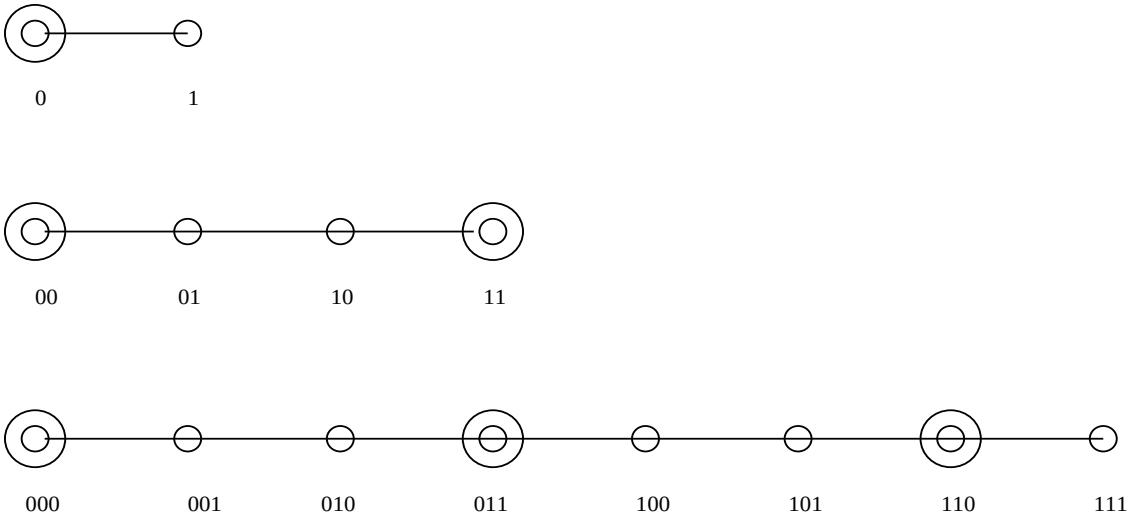
Encoding is the process of mapping the non-negative integers to the set of all codewords. For a given value of n, the number of digits per string, the map is from $\{0, 1, 2, ..., |C_n| - 1\}$ to C_n, where C_n is the set of all codewords for that particular value of n. Decoding is the reverse process of encoding, that is, it is the process of mapping the set of all codewords to the set of non-negative integers, $\{0, 1, 2, ..., |C_n| - 1\}$.

Chapter 2

One-Dimensional Labeling

In the one-dimensional (linear) case, the placement of codewords on the graph is trivial, given our restrictions regarding vertex adjacency and the added restriction that we want the first vertex in the graph to be a codeword. It follows that every third vertex will be a codeword.

The Standard Binary Labeling



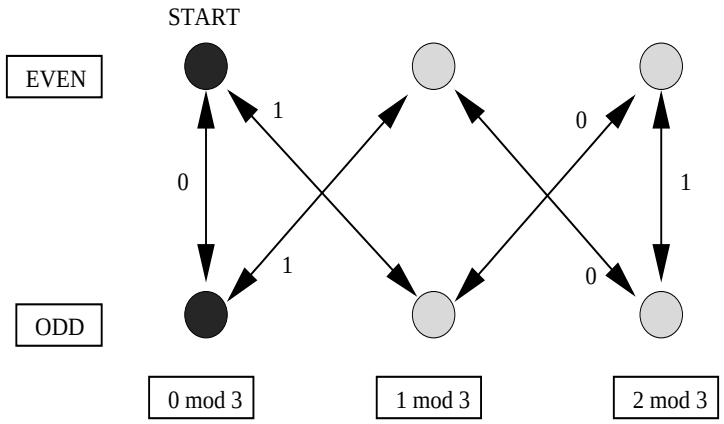
2.1 The Standard Binary Labeling

The Standard Binary labeling is nothing more than labeling the vertices of the graph, in order, in standard binary form. The labelled graph for the $n = 1$, $n = 2$, and $n = 3$ cases is shown on the previous page.

2.1.1 Recognizer

The following figure shows the six-state recognizer for the Standard Binary labeling:

The Six-State Recognizer for the Standard Binary Labeling

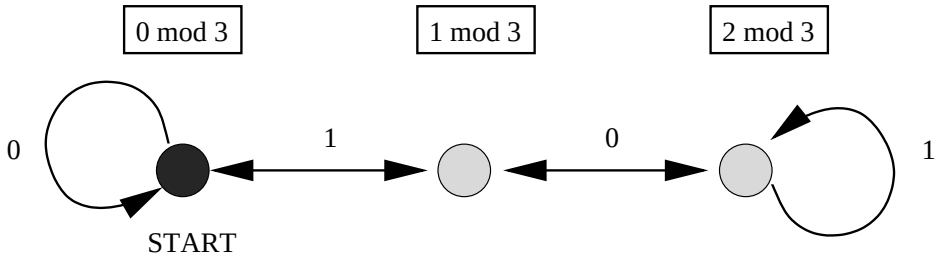


For this machine, as well as all machines in this paper, the labels "EVEN" and "ODD" for rows of states refer to the number of digits read up until the point at which a state in that row is reached. Hence, if a string has and even number of digits, one will end up in a state in the even row. If a string has an odd number of digits, one will end up in a state in the odd row. (Note that as zero is an even number, the starting state is in the even row.) Also, the labels $0mod3$, $1mod3$, and $2mod3$ refer to the position relative to the last codevertex of the vertex corresponding to the string read. For example, observe that the string, 001, is in a $1mod3$ position, the string, 010, is in

a $2 mod 3$ position, and the string, 000, a codeword, is in a $0 mod 3$ position. Also note that the black states are codeword states.

Using a simple algorithm, it can easily be shown that this six-state machine can be reduced to a three-state recognizer. Notice how the symmetry of the six-state recognizer suggests this reduction. More specifically, observe that for all three columns of states, both the even and the odd state have identically marked arrows pointing to states in identical columns. For example, the EVEN-$0 mod 3$ state has its zero arrow pointing to a $0 mod 3$ state, as does the ODD-$0 mod 3$ state, and the EVEN-$0 mod 3$ state has its one arrow pointing to a $1 mod 3$ state, as does the ODD-$0 mod 3$ state. Thus, the two states can be said to be equivalent. Since this is true for all columns, the two rows can be said to be equivalent, and we may "collapse" the six-state machine into the following three-state machine:

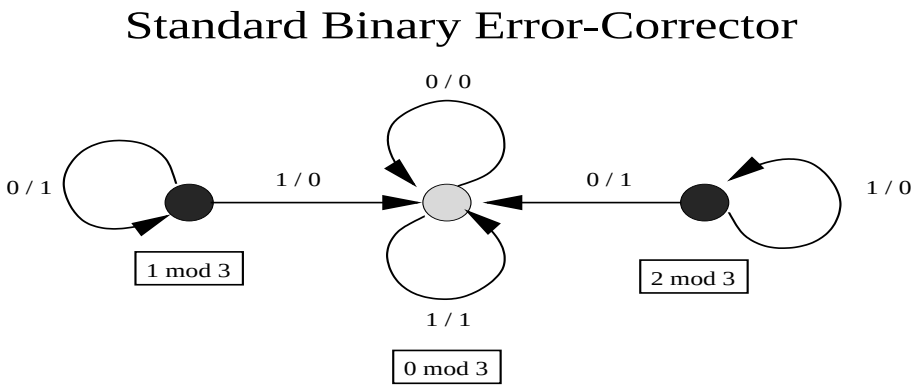
Standard Binary Three-State Recognizer



2.1.2 Error-Correction

Error-correction with the Standard Binary labeling may be done using the following three-state machine. To use the machine, one must first determine the position of the vertex corresponding to the string using the recognizer. Then the string may be read by the error-corrector, starting at the state corresponding to the answer given by the recognizer. Notice that each arrow on the error-corrector is labeled with two digits separated by a slash. On this and all of the following machines, such a two-digit label indicates that

the machine reads the digit on one side of the slash (here, the left side) as an input, as with the recognizer, and the machine will output the digit on the other side of the slash (here, the right side). Thus, as the machine reads the input string one digit at a time, it constructs an output string one digit at a time. With the error-corrector, the output string corresponds to the codeword nearest to the input word. With this machine, strings are read from right to left.



2.1.3 Coding/Decoding

Since the Standard Binary labeling is nothing more than the ordered binary numbering, coding and decoding are relatively easy. To map the non-negative integers to the codewords, one need only multiply the integer by three and write the result in binary form. Thus, in the $n = 4$ case, 0 maps to 0000, 1 maps to 0011, 2 maps to 0110, 3 maps to 1001, 4 maps to 1100, and 5 maps to 1111.

2.2 Labelings with Other Three-State Recognizers

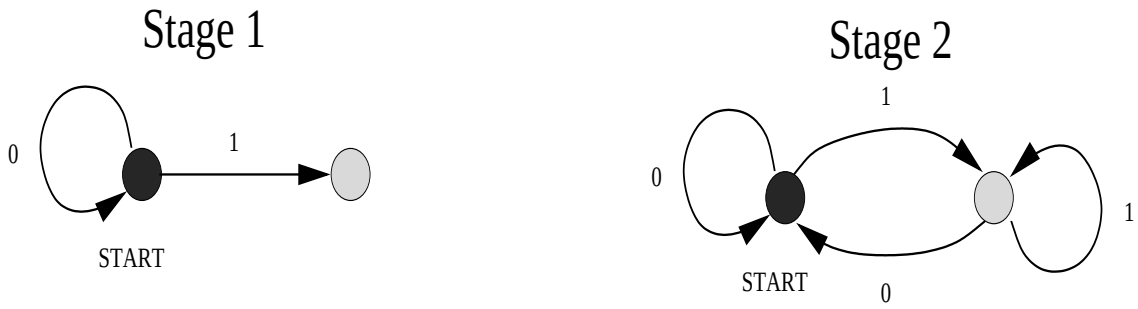
In the search for other labelings as efficient as the Standard Binary labeling, we first examined the three-state recognizer for the Standard Binary labeling

and considered ways of altering it to produce a three-state recognizer for another labeling. We discovered that there are exactly six distinct three-state machines that produce the correct number of codewords and non-codewords when one specifies that for $n = 1$, 0 will be a codeword and 1 will be a non-codeword.

First, we show that at least three states are needed for a recognizer in the one-dimensional case.

Proof First, note that one state is not enough because with only one state, all of the words read will correspond to the same state. Thus, one can not distinguish between codewords and non-codewords. Now consider a machine with two states. Since we would like to distinguish between codewords and non-codewords, we must call one of the two states a codeword state and the other a non-codeword state. Since we have exactly two states, choosing to start at one or the other state will make no difference because the resulting machines will be symmetric either way. Thus, WLOG we choose to start at the codeword state. Since we have chosen 0 to be a codeword and 1 to be a non-codeword in the $n = 1$ case, we must have the zero arrow of the codeword state pointing to the codeword state and the one arrow of the codeword state pointing to the non-codeword state (see figure, Stage 1). We now have two arrows left to insert. For the $n = 2$ case, we need two codewords and two non-codewords. So far, we have 00 as a codeword and 01 as a non-codeword. It is easily seen that to get one more codeword and one more non-codeword, we must have one of the non-codeword state's arrows pointing to the codeword state and the other pointing to the non-codeword state (see figure, Stage 2). This will produce the same number of codewords and non-codewords irrespective of which digits we choose to label these two arrows, so WLOG, we will label them 0 and 1, respectively. Now, we consider the $n = 3$ case and discover that we will have four codewords (000, 010, 100, and 110) and four non-codewords (001, 011, 101, and 111); however, we need three codewords and five non-codewords (see figure of Standard Binary labeling of the graph). Since this machine is the only possible two-state machine which will serve as a recognizer in the $n = 1$ and $n = 2$ cases but does not work for the $n = 3$ case, we conclude that at least three states are needed for the recognizer. QED

Note that for the six possible three-state machines we must have at least one, and no more than two, codeword states in each machine, otherwise the machine would indicate that either all words are codewords or all words are

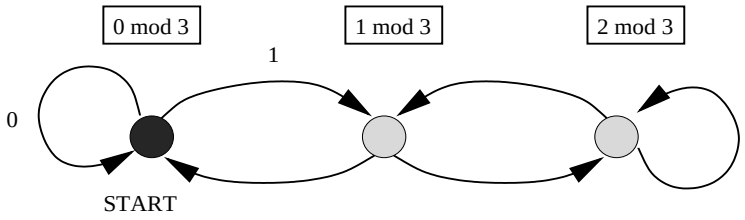


non-codewords.

2.2.1 Recognizers with One Codeword State

Four of the six possible three-state recognizers have exactly one codeword state, one position one state, and one position two state. Thus, these recognizers are capable of distinguishing all three types of words. All of these recognizers have the form shown (see next figure). Notice that the Standard Binary three-state recognizer has this form.

The General One-Codeword-State Three-State Machine

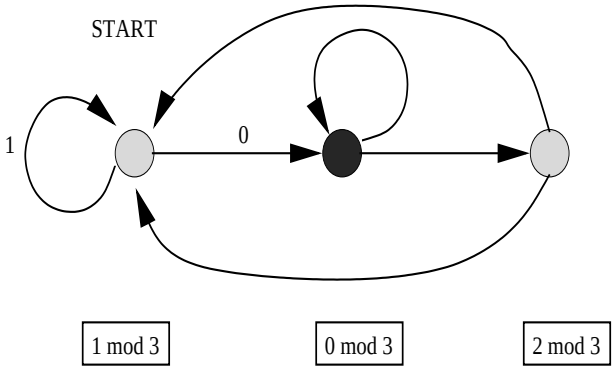


To prove that machines of this form are the only possible one-codeword-state three-state recognizers, we must first establish a starting state in each machine.

Theorem 2.2.1 *For each of the one-codeword-state three-state recognizers, the starting state must be the codeword state.*

Proof Assume that the starting state is the position one state. It is easily seen, by checking the number of words of each type formed for small values of n (in a way similar to that used to show that at least three states are needed), that only machines of the form shown in the next figure will produce the correct number of words of each type up to $n = 3$. If we then arbitrarily assign labels to the arrows and list the words for the $n = 4$ case, we find that in the $n = 4$ case, the machine will produce too many position one words and too few position two words. Thus, this type of three-state recognizer cannot start on the position one state.

Starting at Position One in the One-Codeword-State Recognizer

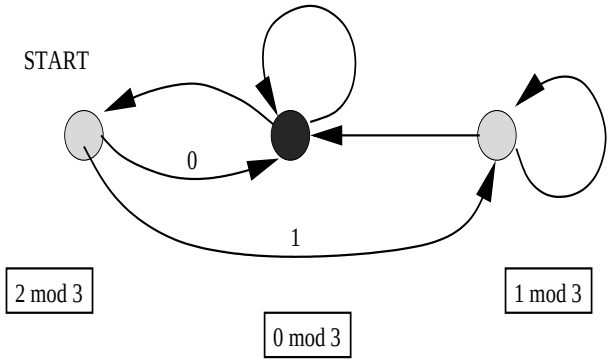


Next, assume that the starting state is the position two state. It is easily seen, by checking the number of words of each type formed for small values of n, that only machines of the form shown above will produce the correct number of words of each type up to $n = 2$. If we then arbitrarily assign labels to the arrows and list the words for the $n = 3$ case, we find that in the $n = 3$ case, the machine will produce too many codewords and too few position one words. Thus, this type of three-state recognizer cannot start on the position two state.

Thus, the starting state in machines of this type must be the codeword state.

Now we are ready to prove that the four one-codeword-state three-state recognizers must have the form of the machine previously shown.

Starting at Position Two in the One-Codeword-State Recognizer

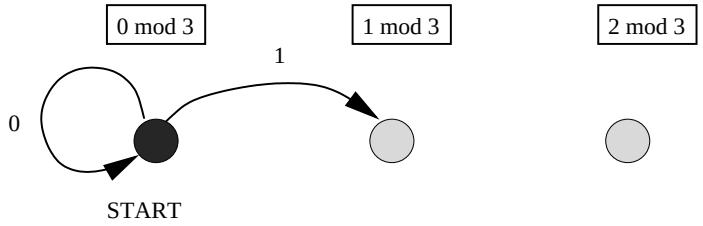


Proof From the previous theorem, we know that the starting state must be the codeword state, and we have decided, WLOG, to have the zero arrow for this state point to the same state and the one arrow point to the position one state. Thus, so far, we have stage one of the machine as shown in the next figure. Now, for $n = 2$, we must have two codewords, one position one word, and one position two word. It is easily seen that for this to happen, the position one state must have exactly one arrow pointing to each of the codeword and position two states, as is shown in stage two of the figure. The only state which remains is the position two state. With little effort, one can see that if this state has any arrows pointing to the codeword state, there will be too many codewords for $n = 3$, and if it has both arrows pointing to the position one state, there will be too many codewords for $n = 4$. Also, if the position two state has both arrows pointing to itself, there will not be enough position one words for $n = 3$. Thus, the position two state must have one arrow pointing to itself and one pointing to the position one state. Thus, the machine must be of the form in the final stage of the figure, which is the form shown previously. We know that machines of this form will work because the Standard Binary three-state recognizer has this form and produces the correct number of words of each type for all values of n. QED

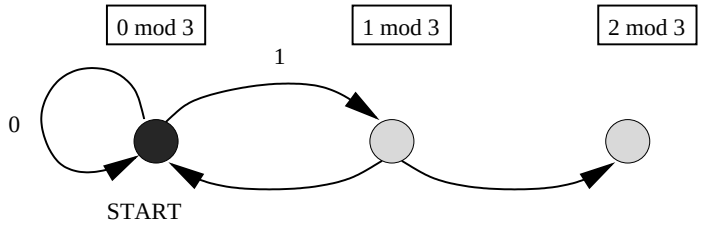
Corollary 2.2.1 *There are exactly four possible one-codeword-state three-state recognizers.*

Proof By the previous theorem, all possible one-codeword-state three-state recognizers are of the form shown in the previous figure. One can easily see that this form allows two possible ways of labeling the set of two arrows for the position one state (seen in parentheses on the figure), and two possible ways of labeling the set of two arrows for the position two state (seen in brackets on the figure). In the figure, one may select either the left digit in both sets of parentheses or the right digit in both sets of parentheses and the left digit in both sets of brackets or the right digit in both sets of

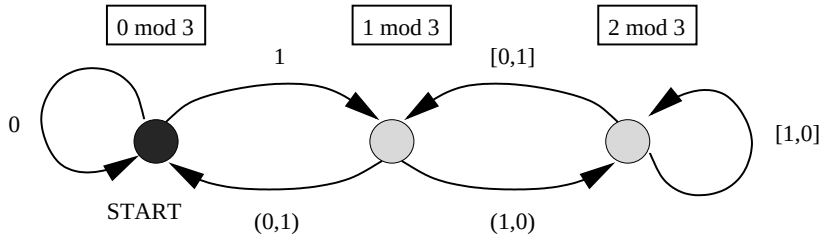
Stage 1



Stage 2



Final Stage



brackets. Notice that the labeling chosen for the position one state's arrows is independent of the labeling chosen for the position two state's arrows. Thus we can multiply the number of possibilities for labeling the position one state's arrows by the number of possibilities for labeling the position two state's arrows to get the total number of possibilities for labeling the arrows of the machine. Thus, we find that there are exactly four possible labelings of the arrows of the machine, and thus, four possible one-codeword-state three-state recognizers. QED

Each of these four distinct recognizers suggests a distinct labeling (one of which being the Standard Binary labeling) for the one-dimensional graph, and each of these labelings will have a simplicity value of 3 for recognition. Thus, at this point, it seems that we have three alternate labelings which are just as efficient as the Standard Binary labeling. We still need to check the other tasks of error-correction and coding/decoding for these other labelings.

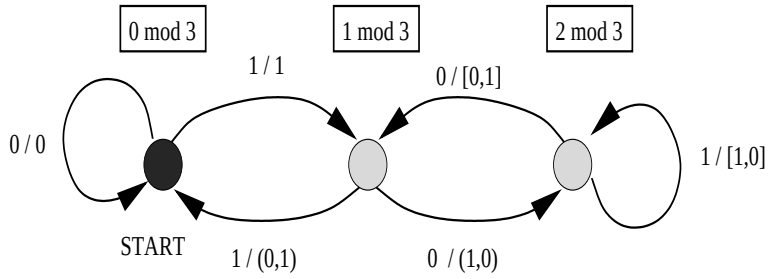
As it turns out, none of the other three labelings readily suggests a finite state method of error-correction or coding/decoding. A nice result, however, is that the labelings for these other machines may be converted to the Standard Binary labelings and vice versa using a three-state finite state machine, which we will henceforth call a converter. Thus, one can take a string from one of the alternate labelings, convert it to a string in Standard Binary, carry out the tasks of error-correction or coding/decoding and convert back to the original labeling all in a finite number of states! Thus, all three alternate labelings are finite state. However, three states are needed every time a string is converted, so it takes more states to carry out the tasks of error-correction and coding/decoding with the alternate labelings than with the Standard Binary labeling. Thus, the Standard Binary labeling remains the most efficient labeling in the one-dimensional case.

The converter for each particular labeling has the exact same form as the recognizer for that labeling, only the arrows are labeled with both inputs and outputs, as with the error-corrector for Standard Binary. For each arrow, the input is identical to the label on the arrow in the corresponding position in the recognizer for the labeling from which one is converting and the output is identical to the label on the arrow in the corresponding position in the recognizer for the labeling to which one is converting. Thus, if you are converting from Standard Binary to alternate labeling "A", for example, the inputs would be the labels of the arrows for the Standard Binary recognizer, and the outputs would be the labels of the arrows for the labeling "A" rec-

ognizer. This construction of the converter makes intuitive sense, since with each digit of one labeling, we would like to change it to a corresponding digit in the other labeling without changing the relationships among the states of the recognizer or the way in which the recognizer reads each digit. With this method of conversion, if reading a digit in one labeling sends one to a specific state of the recognizer for that labeling, that digit will be converted to a digit in the other labeling which will send one to the corresponding state of the recognizer for the new labeling.

The general converter for the one-codeword-state three-state recognizer labelings is shown in the next figure. The arrows are labelled with inputs and outputs. Note that when converting from Standard Binary to an alternate labeling, one reads the left digit as an input and outputs the right digit, and when converting from an alternate labeling to Standard Binary, one reads the right digit as an input and outputs the left digit.

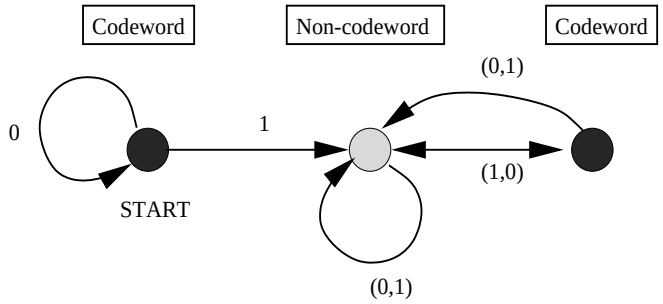
The General Standard Binary-Alternate Conversion Machine



2.2.2 Recognizers with Two Codeword States

The remaining two three-state recognizers have two codeword states and one non-codeword state. Thus, they are not capable of distinguishing between the two types of non-codewords. Nonetheless, these machines do produce the correct number of codewords and non-codewords for all n and are thus worthy of consideration. The general form for these two machines is shown in the figure following the converter on the next page.

The General Two-Codeword-State Three-State Recognizer



To prove that machines of this form are the only possible two-codeword-state three-state recognizers, we must first establish a starting state in each machine, as before. Note that here we have two codeword states and that starting at one codeword state is the same as starting at the other, so we need only show whether the starting state is a codeword state or a non-codeword state.

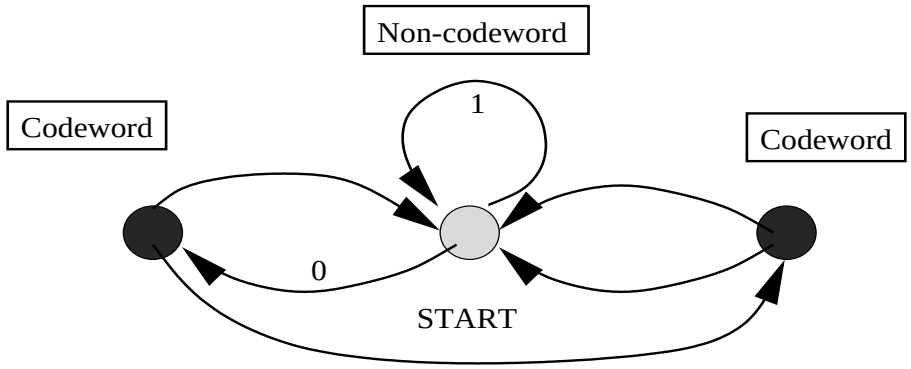
Theorem 2.2.2 *For each of the possible two-codeword-state three-state recognizers, the starting state must be a codeword state.*

Proof First, assume that the starting state is the non-codeword state. It is easily seen, by checking the number of words of each type formed for small values of n, that only machines of the form shown in the next figure will produce the correct number of words of each type up to $n = 3$. If we then arbitrarily assign labels to the arrows and list the words for the $n = 4$ case, we find that in the $n = 4$ case, the machine will produce too many codewords. Thus, this type of three-state recognizer cannot start on the non-codeword state. Thus, recognizers of this form must have one of the codeword states as the starting state. QED

Now we are ready to prove that the two-codeword-state three-state recognizers must have the form previously shown.

Proof We start at one of the codeword states as the previous theorem indicates. We will call this state, state "A", to distinguish it from the other codeword state, which we will call state "B". We see that for the $n = 1$ case,

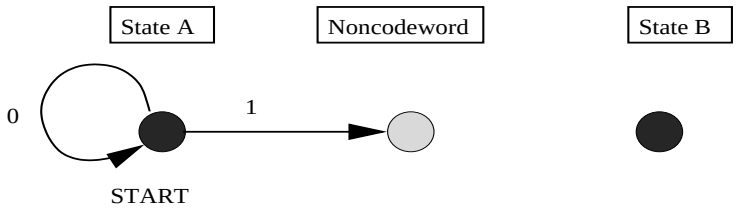
Starting at the Non-Codeword State



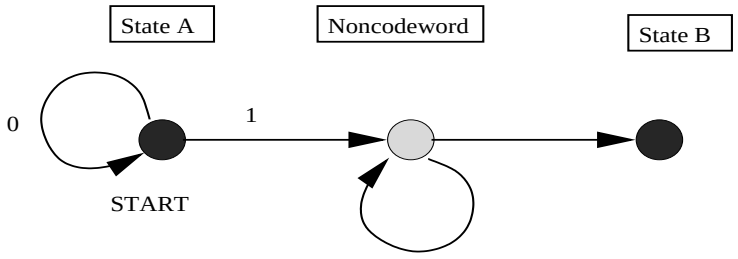
state "A" must have the one arrow pointing to the non-codeword state and the zero arrow pointing to one of the two codeword states. With a little effort, it can be seen that if the zero arrow points to state "B", all resulting three-state recognizers will produce the wrong number of codewords and non-codewords for $n = 2$ or $n = 3$. Thus the zero arrow for "A" must point back to "A" (see Stage 1 of the next figure). Now for $n = 2$, the non-codeword state must have one arrow pointing to itself to provide the correct number of non-codewords. Thus, the non-codeword state's other arrow must point to "B", otherwise we would have only a two-state machine (see Stage 2 of the figure). Finally, we consider the $n = 3$ case and see that in order to have the correct number of codewords and non-codewords, "B" must have both of its arrows pointing to the non-codeword state (see Final Stage of figure). Thus, the machine is completely constructed, and we see that it has the form as shown previously.

It remains to show that this machine will produce the correct number of codewords for all values of n. In the proof for the one-codeword-state three-state recognizer we could simply argue that machines of the form suggested would produce the correct number of words of each type because all machines were of the same form as the recognizer for the Standard Binary labeling, which is known to produce the correct number of all types of words. Here, however, we do not have a labeling known to work which has a recognizer of similar form. Thus, we must prove that the form will work using a differ-

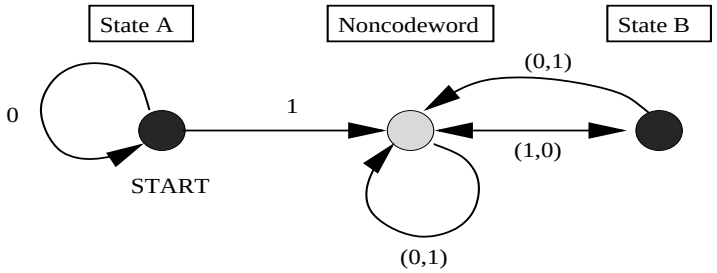
Stage 1



Stage 2



Final Stage



ent method. We will use difference equations to find the number of words produced by each state (i.e. the set of strings which end on each state) of the recognizer for any n, then we will see that the number of codewords and non-codewords produced is correct.

First, note that the total number of words produced for each value of n is 2^n since there are n digits and two possibilities for each digit. Next, observe that for n odd, we would like to have exactly $(2^n+1)/3$ codewords. This fact can be seen by noting that every third vertex in the graph is a codevertex,

starting with the first vertex, and that 3 divides 2^n+1 for odd n. Thus, if we had 2^n+1 vertices for a given odd n, we would have $(2^n+1)/3$ codevertices and the last two vertices of the graph would not be codevertices, but instead we have only 2^n vertices, so we can have the same number of codevertices but will have one less non-codevertex at the end of the graph. For even n, we would like to have exactly $(2^n+2)/3$ codewords. To show this, again look at the graph, noting that every third vertex in the graph is a codevertex, starting with the first vertex, and that 3 divides 2^n-1 for even n. Thus, if we had exactly 2^n-1 vertices for a given even n, we would have $(2^n-1)/3$ codevertices and the last two vertices of the graph would not be codevertices, but instead we have 2^n vertices, so we need to add an additional vertex, which will be a codevertex, since the last two vertices were non-codevertices. Thus, we will have $(2^n-1)/3+1=(2^n+2)/3$ codevertices.

Next, we will set up some convenient notation. We will call the set of words produced by state "A" in case n, A_n, the set of words produced by state "B" in case n, B_n, and the set of words produced by the non-codeword state in case n, N_n. With a little effort, it can be seen that the following equations hold:

$$A_n = A_{n-1} = 1$$

$$N_n = N_{n-1} + 2 * B_{n-1} + 1$$

$$B_n = N_{n-1}$$

Using the easily derived initial conditions for small n and solving these equations yields the following results:

$$A_n = 1$$

$$N_n = (1/6) * [2^{n+2} - (-1)^n - 3]$$

$$B_n = (1/6) * [2^{n+1} - (-1)^{n-1} - 3]$$

With these results, we then continue, noting that $A_n + B_n = C_n$, where C_n is the number of codewords in the n case. We see that $A_n + B_n + N_n = 2^n$

for all n, and that $C_n = (2^n + 1)/3$ for odd n and $C_n = (2^n + 2)/3$ for even n, just as we wanted. Thus, we see that this form of a two-codeword-state three-state recognizer will produce the correct number of codewords and non-codewords for all n. QED

Corollary 2.2.2 *There are only two possible two-codeword-state three-state recognizers.*

Proof First note that there is only one way to label the two arrows from state "A" since we have established that for $n = 1$ the codeword is 0 and the non-codeword is 1. Next observe that both arrows from state "B" point to the non-codeword state, so the two ways of labeling them are the same. Finally, observe that there are two distinct ways of labeling the arrows for the non-codeword state. Thus, there are a total of $1 * 1 * 2 = 2$ ways of labeling the machine. QED

It follows that there are a total of exactly six possible three-state recognizers which produce the correct number of codewords and non-codewords for all values of n. However, notice that since the two two-codeword-state three-state recognizers do not distinguish between the two types of non-codewords, the remaining tasks of error-correction and coding/decoding are likely to be difficult, if not impossible. This conjecture has yet to be proven, however.

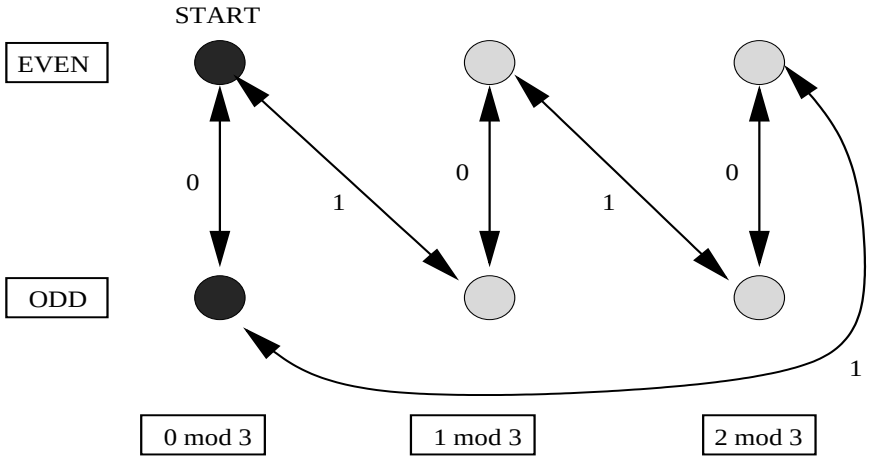
2.3 The Permuted Binary Labeling

Having found all possible three-state recognizers and their labelings and finding no labeling as efficient as the Standard Binary labeling, we next considered the possible ways of altering the error-corrector for the Standard Binary labeling, without altering the recognizer, to see if any other alternate labelings could be found. The initial idea was to use the exact same error-corrector for the Standard Binary labeling, only reading strings from left to right instead of from right to left (as with the Standard Binary labeling). Since the same recognizer would be used for both the Standard Binary labeling and the new labeling, the codewords would be the same for both labelings. Thus, one could use the same method of coding and decoding for both labelings. Since the same recognizer, error-corrector, and coding/decoding methods would be used for both labelings, it was conjectured that the two labelings would be

of the same efficiency. The only difference between the labelings would be the neighborhoods of non-codewords about each codeword.

Upon constructing the labelled graph, for small values of n, using the same ordering of codewords as the Standard Binary labeling but adjusting the neighborhoods according to the error-corrector, it was discovered that in order for the labeling to work, one would need to use an additional two-state machine after using the recognizer and before using the error-corrector to identify whether the word read contained an odd or even number of digits. Only after this information was retrieved could one determine in which state of the error-corrector to start reading the word to correct its errors. Even with this small adjustment, the labeling did not behave well. In particular, for odd n, the vertices of the graph were labelled starting with 00..0 on the left and proceeding to 11...1 on the right, whereas for even n, the vertices were labelled starting with 11...1 on the left and proceeding to 00...0 on the right. It was then discovered that these problems could be fixed by changing the recognizer for the labeling to the six-state machine shown in the next figure. Notice that this machine is essentially the same as the six-state recognizer for the Standard Binary labeling but with the Even-1$mod3$ and Even-2$mod3$ states interchanged, leaving all arrows connected as before.

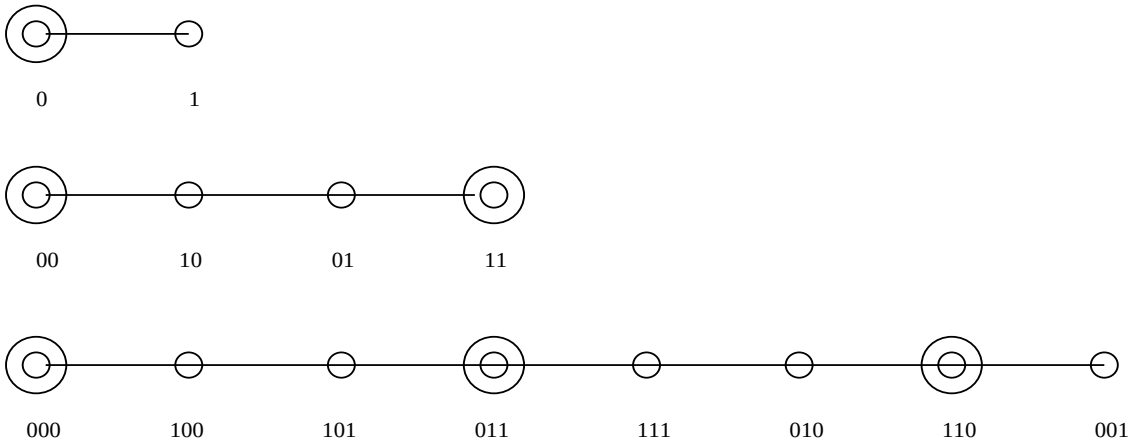
The Recognizer for the Permuted Binary Labeling



The labeling which results has been named the Permuted Binary labeling

since the order of the codevertices is the same as that for the Standard Binary labeling but the rest of the vertices have been permuted. The Permuted Binary labeling is shown in the figure following the recognizer.

The Permuted Binary Labeling



Clearly, the Permuted Binary labeling is not as efficient as the Standard Binary labeling, but it suggests an entirely new class of labelings, one for each six-state recognizer of the same form as the Permuted Binary labeling. Further, these labelings are very close to the Standard Binary labeling in efficiency.

2.4 The Gray Code Labeling

Another well-known and easily defined labeling is the Gray Code labeling. To construct the labelled graph for each value of n, one first takes the labelled graph from the $n-1$ case and copies it twice, the first time forwards, and the second time backwards. Then a zero is added to the left of each of the labels in the first copy and a one is added to the left of each of the labels in the second copy. The labelled graph for the $n = 1$ case is the same as for the Standard Binary labeling. Thus the set of labels, L_n, for the first three values of n are as follows:

$$L_1 = 0 \quad 1$$

$$L_2 = 00 \quad 01 \quad 11 \quad 10$$

$$L_3 = 000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100$$

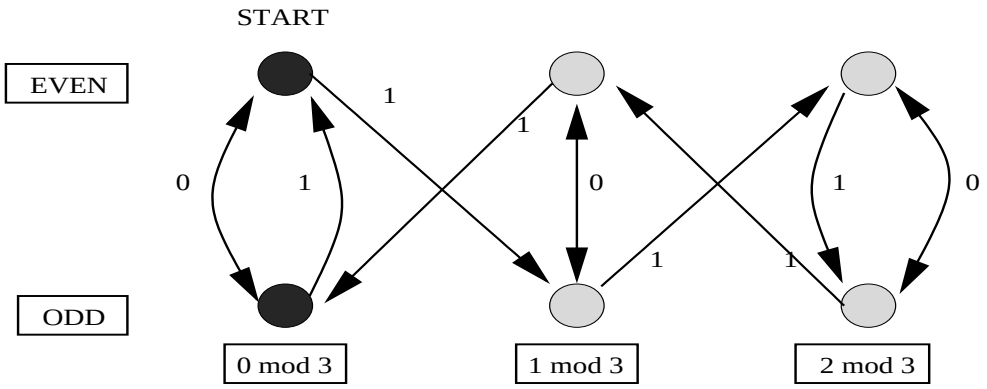
2.4.1 Recognizer

The Gray Code labeling will now be shown to have the following six-state recognizer (see figure on the following page).

Note that for simplicity, in the proof of this recognizer we will refer to the states by the letters E (for even) and O (for odd) to indicate in which row of the recognizer they occur, and we will label these letters with the subscripts 0, 1, and 2 to indicate in which column they occur.

Proof We know that states O_0 and E_0 must correspond to codeword states because a string of only 0's is the first codeword in the graph for any n and we must be able to go back and forth between the odd and even rows using only 0's and still get a codeword every time.

The Recognizer for the Gray Code



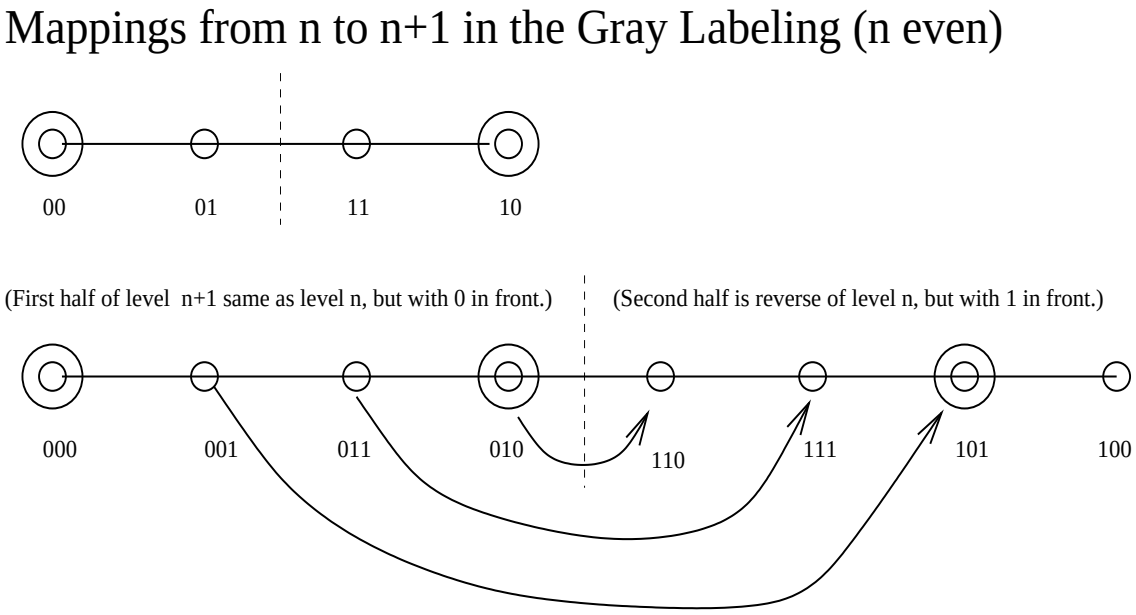
Going from E_0 to O_1, we must have a non-codeword state since 1 is not a codeword for $n = 1$. Also, since 11, 111, and 1111 are all not codewords, O_1, E_1, O_2, and E_2 must also be non-codeword states.

Now, since putting a zero in front of a string from level n to level $n + 1$ means that the new string will be in the first half of the graph, its relative position will remain unchanged, regardless of whether it is in the even or odd case; thus, the zero arrows run back and forth between the following pairs of states: O_0 and E_0, O_1 and E_1, and O_2 and E_2.

Now we must consider what happens when a 1 is placed in front of a string of length n for n even and n odd.

Assume n is even. From earlier, we know that the graph for n even has codevertices at both endpoints. Thus, for level $n + 1$, the second half of the graph must start with two non-codewords. Also, the second half of the graph is in the reverse order of the first half and each word starts with a 1 instead of a 0. The second half, then, is in the reverse order of the graph at level n. Thus, knowing the relative position of a vertex to the last codevertex $mod3$ in level n will allow us to determine the relative position $mod3$ in level $n + 1$ by counting backwards $mod3$ from the end of the graph in level n. We see, then, by looking at the labelled graphs for level n and level $n + 1$, that putting a 1 in front of a string from level n maps a codeword from level n to a position one word in level $n + 1$, a word of position two to a word of position two, and a word of position one to a codeword (see figure). The

arrows on the machine have been labelled according to this rule.



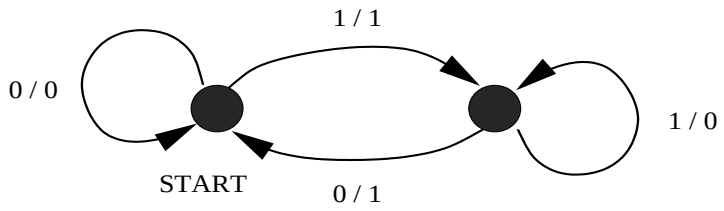
Now assume n is odd. From earlier we see that the graph must have a codevertex followed by a non-codevertex as the last two vertices of the graph. Thus in level $n+1$, in the second half of the graph (where a one is placed in front), we must start with a position two vertex followed by a codevertex, and counting $mod 3$ from the end of the level n graph we see that each word of position one will map to position two, each codeword will map to a codeword, and each position two word will map to a position one word. This corresponds to the one arrows on the machine. Thus the machine shown is an appropriate six-state recognizer for the Gray labeling. QED

2.4.2 Conversion to Standard Binary

The Gray Code labeling does not readily suggest any method of error-correction or coding/decoding, but R.W. Doran provides a method of conversion between the Gray labeling and the Standard Binary labeling [2]. The method of conversion can be accomplished by the two-state conversion machine seen

in the figure. Thus, the tasks of error-correction and coding/decoding are also finite state and only a small degree more difficult than for the Standard Binary labeling.

The Standard Binary-Gray Conversion Machine



Chapter 3

Two-Dimensional Labeling

Labeling in two dimensions occurs on a triangular graph like the one shown in the next figure, which is the Towers of Hanoi labeling. It has been proven that the structure of the graph (i.e. the placement of codevertices) is unique when the top vertex is chosen to be a codevertex [1].

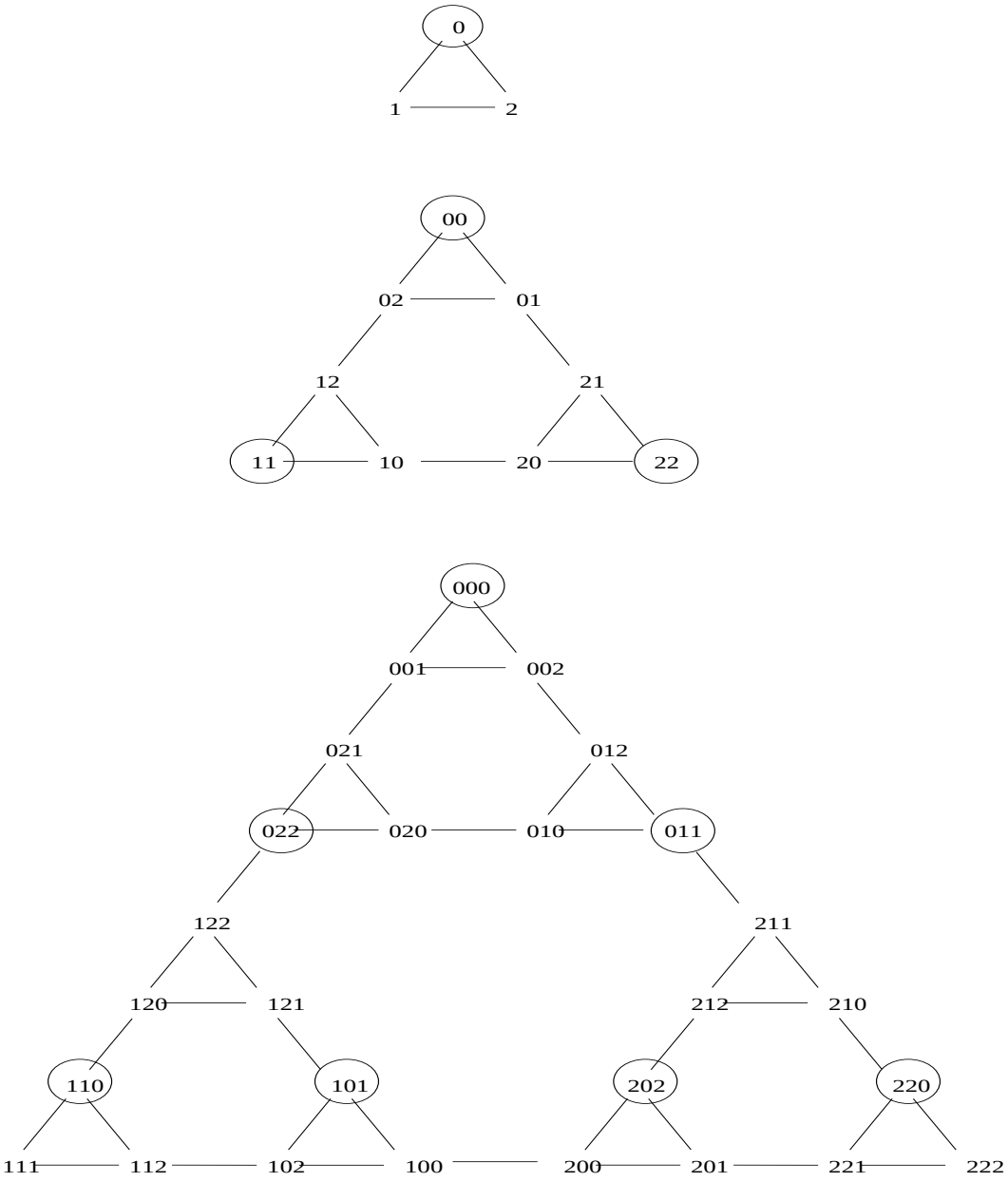
3.1 The Towers of Hanoi Labeling

It has been proven that a finite state labeling for the two-dimensional triangular graph exists [1]. The labeling, as its name suggests is based on the Towers of Hanoi puzzle. With this labeling, each digit of a string corresponds to one of the disks in the puzzle, the right most digit corresponding to the smallest disk and each of the remaining disks corresponding to the disk of the next size larger than the disk corresponding to the digit to the immediate right of the digit being considered. The value of a digit indicates which tower the disk corresponding to that digit is on. Since there are three towers in the puzzle, there are three possibilities for each digit, namely 0, 1, and 2. Thus, the labels of this labeling will be in ternary form. To label the graph, one starts with 000 at the top vertex and then fills in each label based on the legal moves of the Towers of Hanoi puzzle (see [1] for more details on the labeling).

The labelled graph for a given n could more easily be constructed by using the following construction method. First, make three copies of the graph for the $n-1$ case and arrange them in such a way as to make one

larger triangle. Now take the three triangles used to construct the larger triangle and reflect each about its axis which is perpendicular to its base and intersects its top vertex. Now, take the bottom left triangle and rotate it 120 degrees clockwise and take the bottom right triangle and rotate it 120 degrees counter-clockwise. Finally, to the left of all labels in the top triangle, append a zero; to the left of all labels in the bottom left triangle, append a one; and to the left of all labels in the bottom right triangle, append a two. This completes the construction.

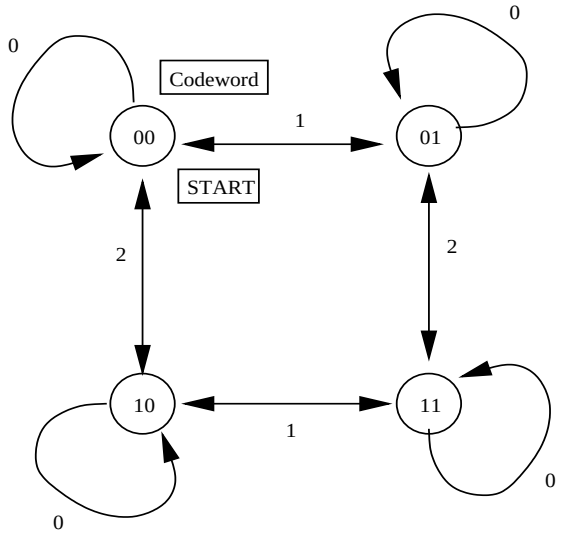
The Towers of Hanoi Labeling



3.1.1 Four-State Recognizer

The Towers of Hanoi labeling is known to have the four-state recognizer shown. This recognizer is based on the parity of each of the digits in the labels. As with the Standard Binary labeling, another finite state labeling exists for each four-state machine of the same form as the Towers of Hanoi recognizer.

The Towers of Hanoi Recognizer



3.1.2 Other Tasks

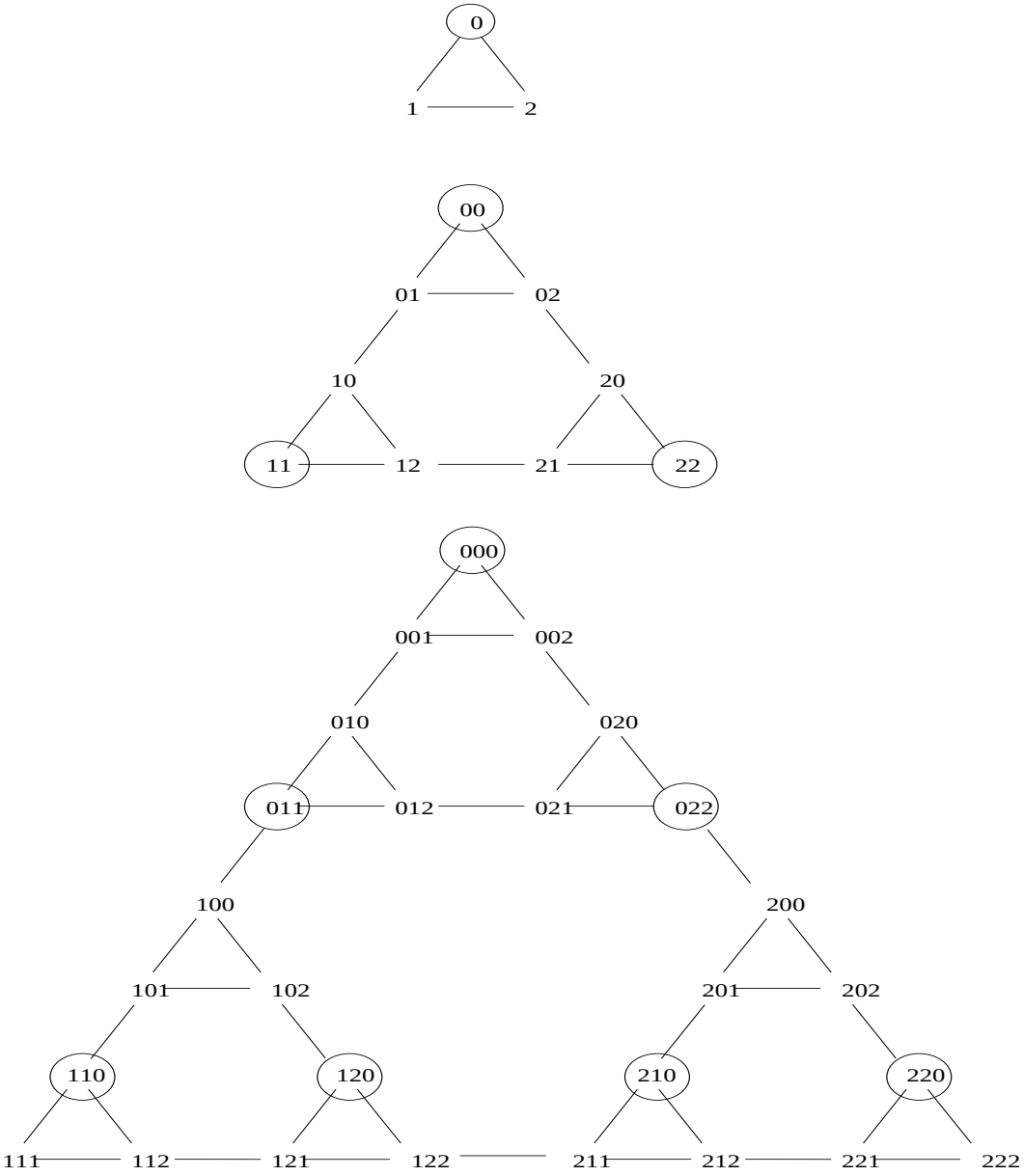
An amazing result is that all of the codewords for the Towers of Hanoi labeling have an even number of 1's and 2's in their labels. This makes the tasks of error-correction and coding/decoding relatively easy. In fact, it has been shown that these tasks can be completed with a finite state machine [1]. Thus, we see that the Towers of Hanoi labeling is a finite state labeling.

3.2 The Ordered (Standard) Labeling

The Towers of Hanoi labeling is fascinating because although it makes little intuitive sense as a method of ordering labels, it still has very easy error-correcting and coding/decoding procedures. The Ordered, or Standard, labeling in two dimensions seems to make more intuitive sense as a way of locating vertices in the graph corresponding to a given string.

The Ordered labeling for a given n is constructed in a way similar to the Towers of Hanoi labeling but without all of the reflections and rotations. First, make three copies of the graph for the $n-1$ case and arrange them in such a way as to make one larger triangle. Now, to the left of all labels in the top triangle, append a zero; to the left of all labels in the bottom left triangle, append a one; and to the left of all labels in the bottom right triangle, append a two. This completes the construction. This method of labeling seems to make a little more sense because one does not need to keep track of rotations and reflections in determining to which vertex of the graph a word corresponds. The digits reveal the exact location of the vertex on the graph. The Ordered labeling of the graph can be seen in the next figure.

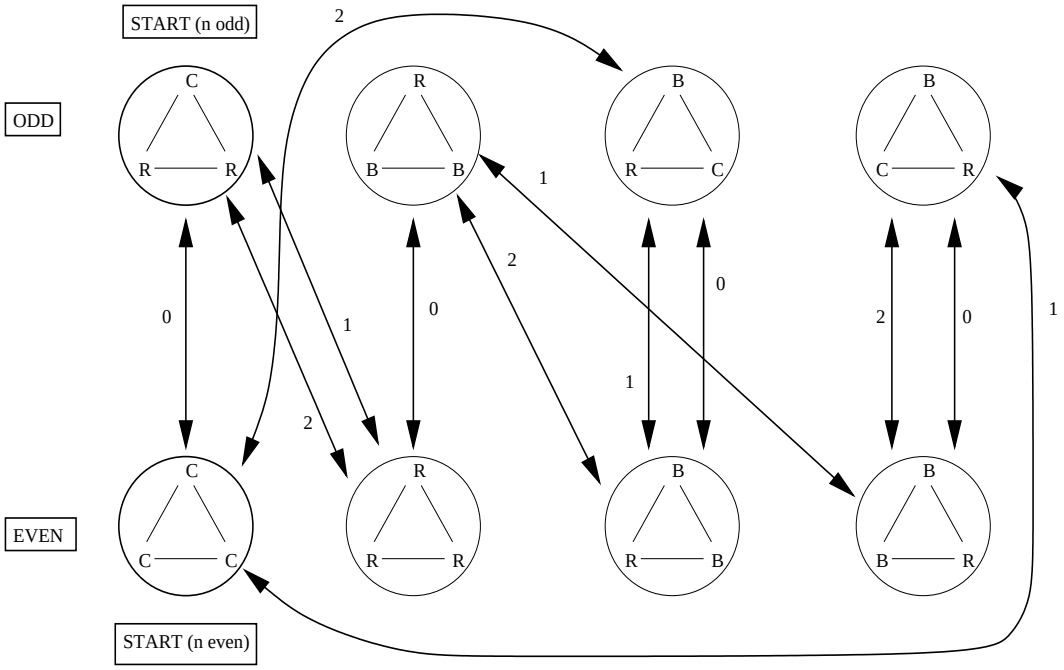
The Ordered (Standard) Labeling



3.2.1 Recognizer

As you can see from the construction of the ordered labeling, for a given n, the trianglular graph consists of three sub-triangles,each of which, in turn, consists of three smaller triangles. Cull and Nelson showed that when the vertices of each of these triangles are labelled with R's, C's, and B's based on there relationship to codevertices, the triangles can be arranged in only a small number of ways [1]. One can use this knowledge to determine to which vertex in the graph a given string corresponds. Thus, a finite state recognizer may be constructed for the Ordered labeling. The machine shown in the figure was found to work as a recognizer for the Ordered labeling. The two starting states are also codeword states.

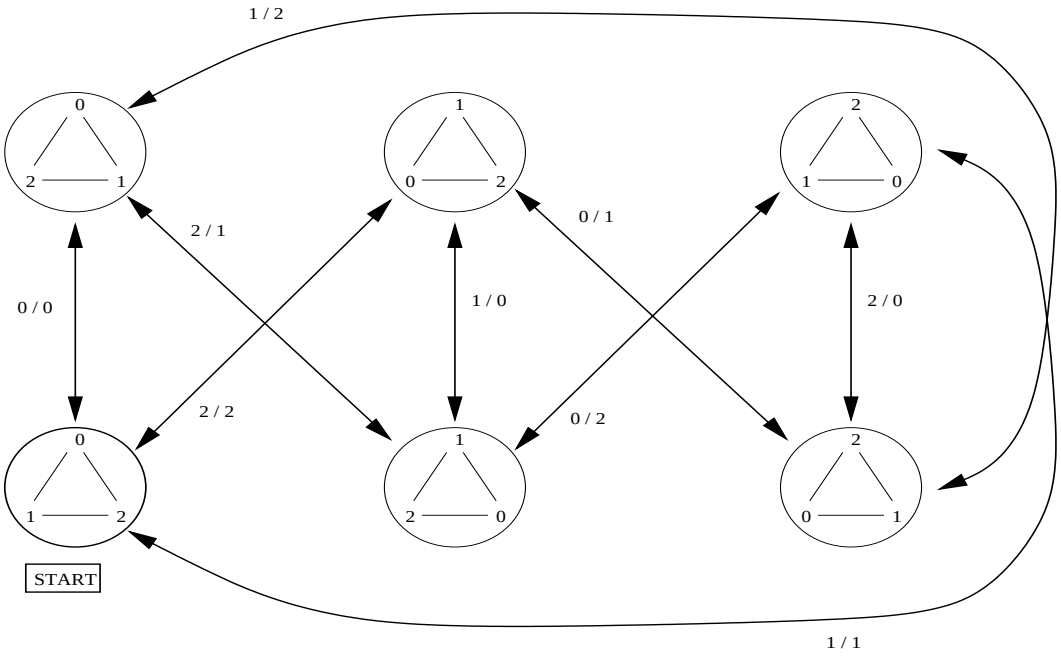
The Ordered (Standard) Recognition Machine



3.2.2 Conversion to Towers of Hanoi

Although it seemed that the Ordered labeling would provide an easy method of error-correction and coding/decoding based on its sensical construction, no easy methods presented themselves readily. In the search for an error-corrector and coding/decoding machine, however, it was discovered that a finite state conversion machine exists that will convert back and forth between the Ordered labeling and the Towers of Hanoi labeling. Thus, it was found that the Ordered labeling is also finite state, though not as efficient as the Towers of Hanoi labeling. The conversion machine can be seen in the figure.

The Towers of Hanoi-Standard Conversion Machine



We now prove that the conversion machine shown will work. First, we will set up some convenient notation. We will call a triangle of any size in the labelled graph an "a-b-c triangle" if the triangle has a string ending in the digit "a" at its top vertex, a string ending in the digit "b" at its bottom

left vertex, and a string ending in the digit "c" at its bottom right vertex. Now note that there are exactly six distinct types of triangles, because we have exactly three distinct digits (0, 1, and 2) and there are exactly six permutations of these digits (0-1-2, 0-2-1, 1-0-2, 1-2-0, 2-0-1, and 2-1-0). These six types of triangles can be seen in the states of the converter. Now, note that if we can read a word from the Towers of Hanoi labeling digit by digit and know exactly in which type of triangle the vertex corresponding to the string must be after each digit, then since in the Ordered labeling we know that every string is in a 0-1-2 triangle due to the graph's construction, we can set up a conversion between the two labelings.

Claim 3.2.1 *In any a-b-c triangle of the Towers of Hanoi labeling, if the next digit read (from the left) is "a", then the next triangle considered will be an a-c-b triangle; If the next digit read is "b", the next triangle will be c-b-a; and if the next digit read is "c", the next triangle will be b-a-c. Further, the largest triangle is a 0-1-2 triangle.*

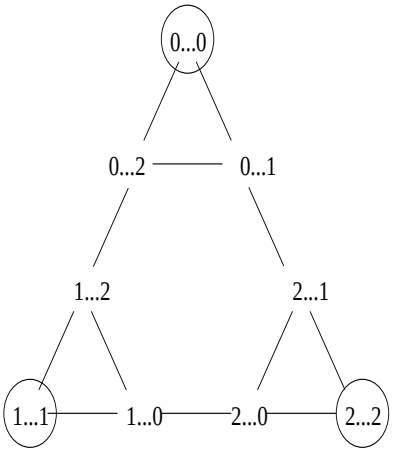
Proof Notice that for $n = 2$ the claim is true (see Towers of Hanoi Labeling figure). We start looking at a 0-1-2 triangle. If the first digit read is 0, we are in a 0-2-1 triangle; if it is 1, we are in a 2-1-0 triangle; and if it is 2, we are in a 1-0-2 triangle.

Now assume the claim holds for the $n = k$ case. The $k + 1$ case will consist of three copies of the k case triangle, only reflected and possibly rotated. Notice that reflections and rotations act not only on the vertices of the copies of the k case triangle, but also on all of the smaller interior triangles which make up this outer triangle since the inner structure is rigid. Thus, though the labels may be in new places, the relationships among them will remain fixed. Thus the claim will still hold for the smaller triangles.

It remains to show that the $k+1$ triangle, the largest triangle, will behave according to the claim.

Note that the k triangle is a 0-1-2 triangle and that by the construction of the graph, we will reflect it to make the top triangle of the $k + 1$ triangle, reflect and rotate it 120 degrees clockwise for the bottom left triangle, and reflect and rotate it 120 degrees counter-clockwise for the bottom right triangle in the $k + 1$ triangle. Further, we will append a 0 to the left of all strings in the top triangle, a 1 to the left of all strings in the bottom left triangle, and a 2 to the left to all strings in the bottom right triangle. Thus,

the new triangle will have a 0-2-1 triangle on top, a 2-1-0 triangle in the bottom left, and a 1-0-2 triangle in the bottom right. We see that the large $(k+1)$ triangle is a 0-1-2 triangle and that reading a leading 1 puts us in a 2-1-0 triangle, reading a leading 2 puts us in a 1-0-2, and reading a leading 0 puts us in a 0-2-1 triangle (see figure). Thus, the claim holds for $n = k+1$. Thus, by induction, it holds for all n. QED



Using the claim, we can build the converter, sending arrows to the states corresponding to the types of triangles considered after reading each digit. Now we would like to put in the corresponding Ordered labeling digits on each of the arrows. We know that in the Ordered labeling, all triangles are 0-1-2 triangles. Thus for any triangle, a-b-c, in the Towers of Hanoi labeling, "a" corresponds to 0, "b" corresponds to 1, and "c" corresponds to 2 in the Ordered labeling. Thus, for any state of the converter, we can look at the labels of the vertices of the triangle in that state and match them to their corresponding digits in the Ordered labeling; then we can see to which arrows the labels of the vertices of the state's triangle correspond and take the corresponding digits of the Ordered labeling and assign them to the same arrows. Thus, we have our converter.

Conclusion

In our search for alternate labelings of the one-dimensional and two-dimensional graphs which support perfect one-error-correcting codes, we discovered that for each finite state labeling, there is an additional finite state labeling for each finite state machine of the same form as the recognizer for the given labeling. Each of these new labelings may be converted to the given labeling by a machine with the same number of states as the recognizer. Thus, there is not a unique labeling with a minimal state recognizer in any dimension. In fact, there are many labelings with a minimal state recognizer. Further, there may be many other finite state labelings, in a given dimension, that have a recognizer with more than the minimal number of states or with a different form from the given labeling's recognizer. These labelings may also be convertible to the original labeling, though this is not necessarily true. Other alternate labelings may be produced by making minor alterations in the recognizer, error-corrector, or coder/decoder for a given labeling.

Due to the existence of these alternate labelings, the Standard Binary and Towers of Hanoi labelings are not unique finite state labelings, nor are they unique labelings with a minimal recognizer. However, no labeling has been produced that is as efficient as or more efficient than the Standard Binary labeling in one dimension or the Towers of Hanoi labeling in two dimensions on all three tasks. Thus, these two labelings may be the all-around most efficient labelings in their respective dimensions. Further research is needed to prove that this is true.

Other research in this area may be directed to finding the number of labelings of a given efficiency. Counting the number of possible recognizers for a given number of states may help in completing this task. Additional research may be directed to considering the different ways in which recognizers, error-correctors, and coders/decoders can be altered to produce alternate labelings. Finally, looking at alternate labelings in higher dimensions may be another avenue for continued research.

References

[1] P. Cull and I. Nelson. *Error-Correcting Codes on the Towers of Hanoi Graphs.* Technical Report-NSF Grant DMS 93-00-281. Department of Computer Science, Oregon State University. Corvallis, Oregon. August 1995.

[2] R.W. Doran. *The Gray Code.* Technical Report-131. Department of Computer Science, The University of Auckland. Auckland, New Zealand. 1997