Finite State Automata and the Tower of Hanoi

The first time you were introduced to functions, your instructor may have described a function as a machine that, when given an input value, produces an associated output value. Let's actually design a "function machine," called a *transducer*. A transducer is a kind of computer, or *automaton*, that behaves according to encoded instructions.

For most of this chapter, a transducer is able to read a finite binary string, one character at a time, reading from left to right. The domain of a transducer will consist of finite binary strings; **not** the binary numbers represented by these strings, but the strings themselves. For example, the string 00 is a different string than the string 000. In Section 1, we will consider transducers that read ternary strings.

We begin by listing the general form and requirements of a *deterministic finite-state transducer* which we will describe in more detail immediately after:

- It has a finite number of *states*.
- One of its states is designated as its *initial state*.
- It can read only a defined finite set of characters, called the *alphabet*, one at a time.
- Its input strings must be of finite length.
- Given a state and any possible character, a set of instructions determines specific responses (this is the *deterministic* part of its name).
- It can read that the string has come to an end, at which point it enters its initial state and halts.

The states help determine how the transducer behaves. As the string is fed in the transducer is configured in its initial state (indicated in a diagram by dashed lines), and it reads the first character. When the transducer reads a character, it responds in two ways: 1) it prints out a binary character; and 2) it enters a new state. It then reads the next character of the string and responds to it. When the last character is read, the transducer returns to its initial state. It has finished printing the output associated with the given input. Note that the length of the output string always equals the length of the input string.

We will start with an extremely simple function machine, A_z, called the "Zero Transducer." The function rule is going to be: for any character of the given input

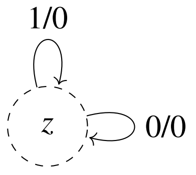


Figure 1: *The Zero Transducer automaton A_z*

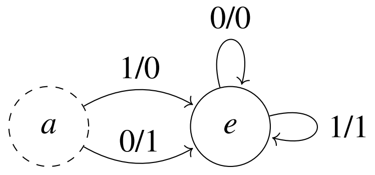


Figure 2: *The transducer A_a for automorphism a*

string, print out the 0 character. See Figure 1 for a diagram of this transducer. The circle labelled z represents the initial (and only) state, and the $1/0$ at the top instructs the transducer, or automaton, to print a 0 when it reads a 1, while the arrow directs the automaton to return to state z; the $0/0$ on the right instructs the automaton to print a 0 when it reads a 0, while the arrow directs the automaton again to return to state z. For example, when we feed the string 101 into A_z, moving left to right, A_z changes 1 to 0, leaves 0 alone and then changes 1 to zero to yield 000 as the output.

Although A_z serves as a simple example of a transducer, the function produced is not one we are interested in here because it does not act as an "automorphism machine." In this context, we are interested in transducers which represent automorphism machines. If we want our automata to act as automorphism machines, their encoded rules must produce a function that is a bijection whose domain is the same as its range, so the strings that are output must come from the same set of strings that are allowed as input. The Zero Transducer does not produce a bijection, so it is **not** an automorphism.

Figure 2 shows a transducer called A_a that requires two states to accomplish its function assignment, which is to change the initial character (from 0 to 1, or 1 to 0), leaving the remaining input characters the same.

For example, when we feed the string 101 into A_a, moving left to right, 1 is changed to 0, the transducer enters state e (the identity state), and the rest of the string remains the same, producing an output of 001.

Although it is possible to define transducers with an infinite number of states, we will work here only with finite-state transducers and only a finite number of transducers at a time. The consequence is that all the groups generated by our automata will be finitely-generated. There are also different types of automata than the transducers we have described, but here we will work only with transducers; hence, when we refer to “automata” without further explanation, we will mean “deterministic finite-state transducers.”

Figure 3 is an example of the automaton representing a familiar group, the Lamplighter Group L_2.

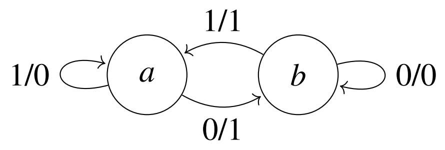


Figure 3: *The Lamplighter automaton A*

1 Tower of Hanoi

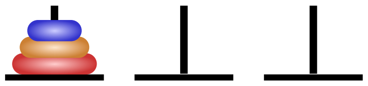
Traditionally, the Tower of Hanoi game is played with three pegs and any number of disks of different sizes which fit onto any of the pegs. In this section, we will focus on a game played with three disks. The disks are initially stacked on the first peg in size order with the smallest on top. The object of the game is to move the stack from the first peg to another, retaining the size order. The rules for moving disks are:

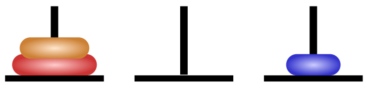
1. Only one disk can be moved at a time, from the top of a stack, to any other peg, with the following restriction:

2. No disk may be placed on top of a disk smaller than itself.

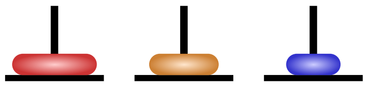
Figure 4 shows the optimum sequence of moves to win the game played with three disks.

The game was invented in 1883 by E. Lucas, a French mathematician, who may have been inspired by a legend that may have originated in Vietnam or India. The legend describes a room in a temple with three posts and 64 golden disks,

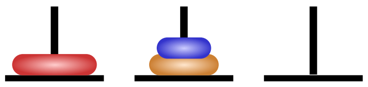




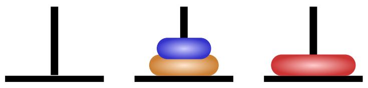
Moved disk from peg 1 to peg 3.



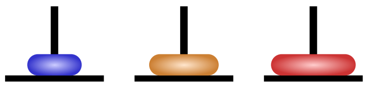
Moved disk from peg 1 to peg 2.



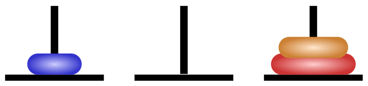
Moved disk from peg 3 to peg 2.



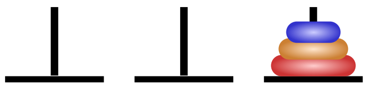
Moved disk from peg 1 to peg 3.



Moved disk from peg 2 to peg 1.



Moved disk from peg 2 to peg 3.



Moved disk from peg 1 to peg 3.

Figure 4: *The Tower of Hanoi played with 3 disks*

each of different size. Monks are tasked with moving the disks according to the above rules. It is said that when the task is completed, the world will come to an end.

It is well-known that to complete a Tower of Hanoi game with three pegs and n disks, the shortest number of moves required is $2^n - 1$. For the 64-disk game, moving the disks at the rate of one per second will take about 585 billion years.

Since the game is easy to learn, yet challenging to complete, it has made many appearances in popular culture. In the 1959 science fiction movie "Now Inhale," the Earthling hero is allowed to play one game from his home planet before he is executed. He chooses to play the Tower of Hanoi (presumably with many disks). In 1966, it appeared as a 10-disk game in a "Dr. Who" episode. In the 2011 movie "Rise of the Planet of the Apes," the game is used as an intelligence test for captive apes. It has also appeared in the television show "Survivor" and one of the Star Wars movies, among others.

An automaton can be built to play the Tower of Hanoi for a given number of pegs, which can accommodate any number of disks. If we wish to change the number of pegs, a new automaton needs to be built, with a different number of characters in the alphabet and a different number of states.

Here, we will consider the automaton for the traditional version, using three pegs. A summary of ingredients necessary for the 3-peg, 3-disk game is:

- three pegs, labelled 1, 2, and 3
- three nontrivial states a_{ij} $(i \neq j)$ in the automaton, each governing the possible moves between peg i and peg j
- a 3-string giving the current position of each of the three disks
- three disks, named s, m, l.

To identify the positions of the disks on the three pegs, we need an alphabet of three characters; hence, the automaton H_3 (see Figure 5) represents automorphisms of the infinite complete rooted ternary tree. Each state in the automaton represents a possible move between one peg and another, and we name the states accordingly: state a_{12} as the initial state governs possible disk moves between peg 1 and peg 2, in either direction; using a_{23} governs a disk moving between pegs 2 and 3; and using a_{13} governs a disk moving between pegs 1 and 3.

Notice that each state is unrelated to the others. They do not refer to each other, only e, which makes sense because only one disk can be moved at a time. The

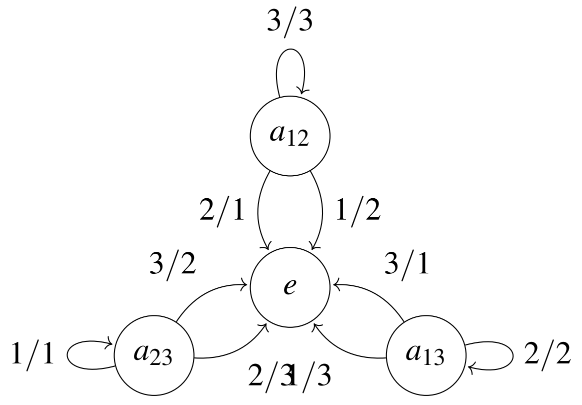


Figure 5: *Automaton of H_3*

automaton is configured so that you cannot make a "wrong move" (big disk on top of smaller disk, or moving a disk not at the top of the stack), but you can make moves that create unproductive steps. Whenever a disk is going to be moved, an initial state in H_3 must be chosen. Only one choice of initial state will move the game toward completion.

The number of disks is represented by the length of an input string. H_3 works for any number of disks; but once the number of disks is fixed at k, so is the length of the strings. In our game we will use three disks, so every string must have length three. The character in the first position represents the location of the smallest disk (on peg 1, 2, or 3); the character in the second position represents the location of the middle-sized disk; and the character in the third position represents the location of the largest disk.

Example 1. Let's begin the game, with the three disks (*s, m, l*) stacked in size order on peg 1. Refer to the automaton in Figure 5. The string that represents this configuration is 111. The object is to move all the disks to peg 3.

Suppose we choose a_{23} as the initial state, which is the state that governs moves between peg 2 and peg 3. Since all the disks are on peg 1, there are no possible moves between peg 2 and peg 3, and the input string 111 does not change; therefore, as a "move," it is unproductive. However, starting with a_{13} as the initial state, disk *s* moves to peg 3 and the output string is 311.

The next move begins with choosing an initial state for the input 311, preferably avoiding unproductive moves. If we choose correctly, we can complete the game in 7 moves.

◇

Once the number of disks to use is fixed at k (hence, the length of every string is fixed at k), we are only interested in permutations of the ternary rooted tree at level k. Thus, a Schreier graph is extremely useful. Figure 6 shows the Schreier graph for H_3 at level 3, which is a far more useful tool in studying the game than the automaton. Notice how easy it is to find the shortest path from the string 111 (representing all disks on peg 1) to the string 333 (representing all disks on peg 3).

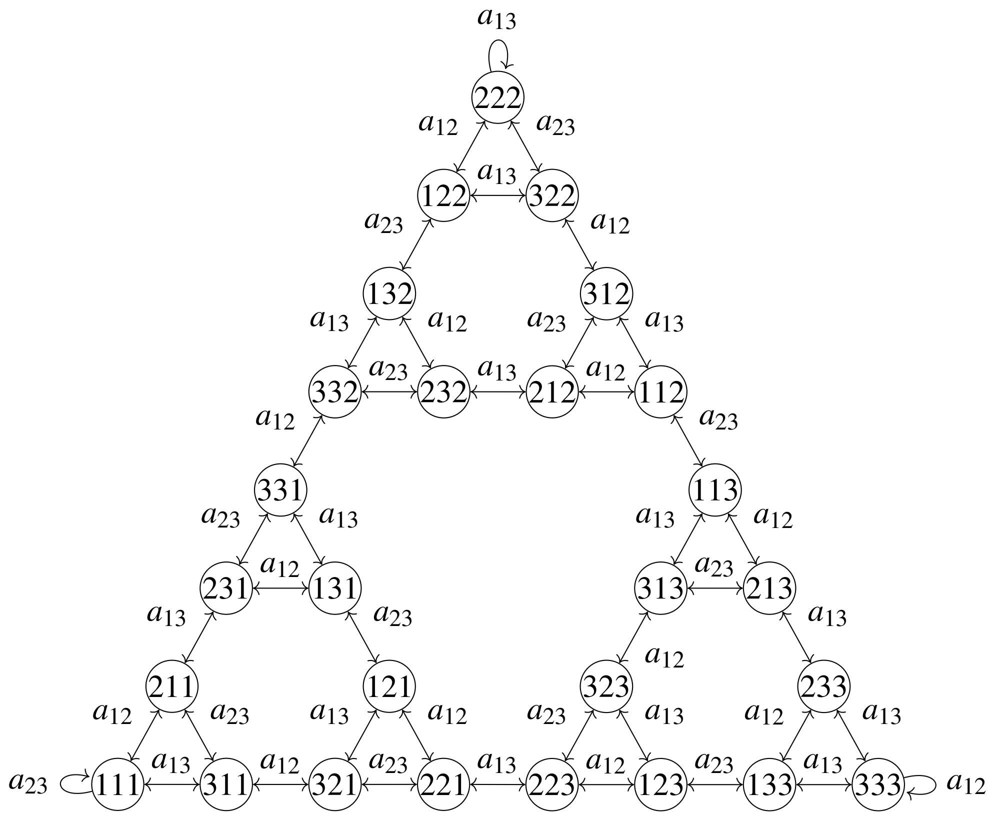


Figure 6: *Schreier graph of H_3 for level 3*