Robin Weiss
4.20.2009
COMP261

Cellular Automata: History, Theory, and Turing Machines

This paper will begin with a brief history and background of the concept of cellular automata, but will then focus on a particular example -- John Conway's Game of Life. We will explore some of the interesting patterns found in this system and eventually show how one can construct a fully functional Turing machine from patterns in this cellular automaton. The final sections of this paper will explore other applications of cellular automata and reflect on the constructions presented herein.

Cellular Automata

Cellular automata have their origin in theoretical systems described by John von Neumann and Stanislaw Ulam in the 1940's. The collaboration of von Neumann, considering the notion of self-replicating robots, and Ulam, exploring patterns in crystal growth, would lead to the first formal description of a cellular automaton.

This area of study, while relatively young, has proven useful in modeling natural processes. We find time and again that while the basic physical laws of everyday processes are well-known, the intricate ways in which discrete components of a system interact to create highly-complex global behavior have defied even qualitative analysis. (Wolfram 1983) What is needed in moving forward is a system that captures the nature of interactions between discrete components of some universe. Cellular automata are a good candidate for such a system and some specific applications are examined in a later section of this paper.

Cellular automata are, by definition, dynamic systems that operate in discrete time and space where global behaviors of the system are characterized by the local interactions of the parts. In general, cellular automata are described by a 2-tuple, {C , R}, where C is a collection of "cells" and R is a set of "rules" for how each cell behaves. At time step *t*, each cell will take on one of a finite number of states. The rules of a cellular automaton describe the state of each cell at time step $t + 1$ based upon some

characteristic(s) of a cell's neighbors (cells geometrically adjacent to the one in question).

To make the description concrete, we consider here a very well understood and widely studied instance of cellular automata known as The Game Of Life (GOL). Invented by John Conway and popularized by a feature article in Scientific American magazine in the 1970's, GOL is simple yet interesting example of a two-dimensional cellular automaton. Cells in the automaton take one of two states, "alive" or "dead" (indicated by the colors black and white respectively), and each cell updates its state based upon the states of its 8 closest neighbors. The neighborhood of a cell is shown in figure 1 below.

The state of each cell in GOL is determined by the following rules:

For a cell that is "alive":
- If cell has one or zero live neighbors cell dies
- If cell has four or more live neighbors cell dies
- If cell has two or three live neighbors cell survives

For a cell that is "dead":
- If cell has three live neighbors cell becomes alive
- Else, cell remains dead

It should be noted that each cell essentially functions as a finite state automaton, and as such, we can provide a transition diagram for each cell (shown in figure 2).

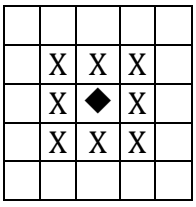

Figure 1: X's denote the neighbors of the cell at the center of the grid marked by ◆

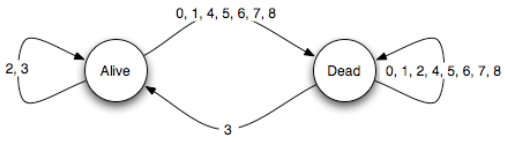

Figure 2: The transition diagram for cells in The Game Of Life

The rules given above create complex patterns in the grid where groups of live cells change configurations in both very predictable as well as seemingly random ways. The predictable patterns found in this cellular automaton will be further explored in the

next section of this paper as we investigate how live cells can be arranged to simulate the computation of arbitrary Boolean logic functions.

The Cellular Automata-based Turing Machine

This section will explore special arrangements of live cells in GOL which can be interpreted to represent data as well as Boolean logic functions. These constructs enable the construction of a Turing machine in the space-time dimension of the cellular automaton. The proof is by construction and is based on the fact that we can provide an abstraction from cells to data, and from data to the 3 fundamental Boolean logic operators – AND, OR, and NOT. With this, it becomes evident that one could construct a general purpose computer equivalent in power to a Turing machine.

In order to give a construction of a general purpose computing device in GOL we must first introduce some fundamental objects.

Gliders

Gliders are an example of a mobile pattern, one that every 5 generations completely replicates itself with an offset of one cell in both the vertical and horizontal directions (see figure 3 below). Gliders can of course be constructed to move in any direction by simply rotating and/or reversing the arrangement of cells.

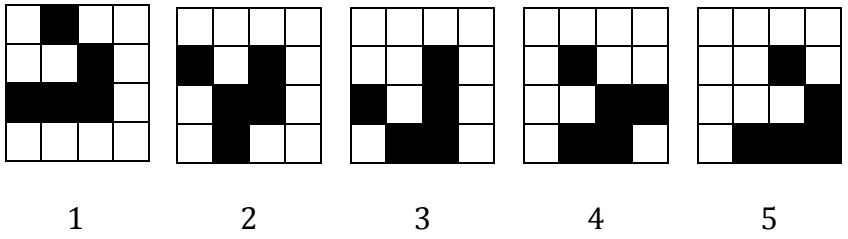

Figure 3: The replication of a single glider over 5 generations

An interesting and useful property of gliders is that when a head-on collision occurs, both gliders annihilate. Figure 4 shows a collision between a right-flying glider and a left-flying glider.

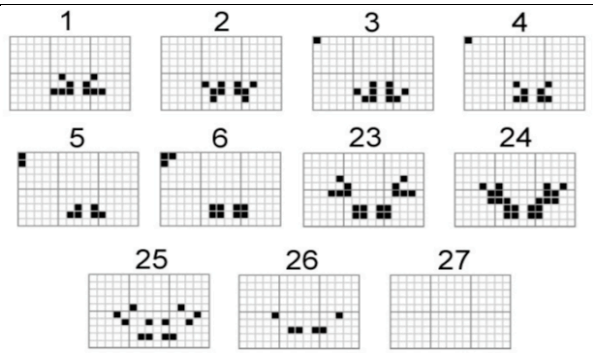

Figure 4: The head-on annihilation of two gliders (Rennard 2002)

The annihilation process is two-phase. First, the collision generates two 2 x 2 blocks. When the next two gliders collide with the blocks, the gliders and the blocks are annihilated completely.

Glider Guns

An other necessary component is a pattern that can generate gliders. These patters are dubbed glider guns and are able to periodically generate gliders. One of the most useful of these patterns is the p30 glider gun shown in figure 5.

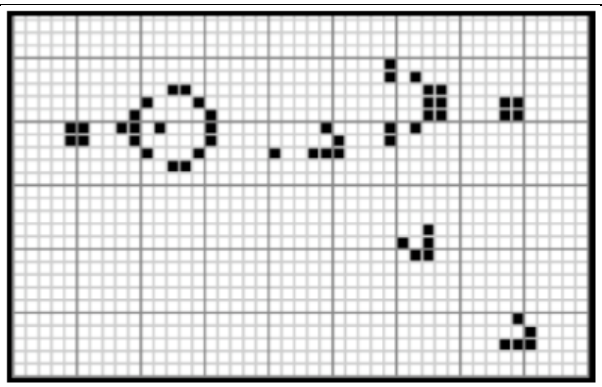

Figure 5: The p30 glider gun. Gliders are seen flying to the right. (Rennard 2002)

The p30 glider gun has the property that every 30 generations a glider is generated. Glider guns can be constructed to shoot gliders in any direction by rotating and/or reversing the arrangement of cells.

Stoppers

Also necessary are devices that can stop a glider stream. So called stoppers are immobile, stable patterns that upon a collision with a glider remain in tact but annihilate the glider. One such pattern is shown in figure 6 below.

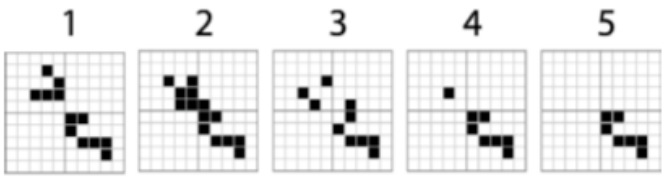

Figure 6: In generation 1, a glider (top left) is about to collide with a stopper (bottom right)

With these objects, we can now explore how they can be used to represent data and subsequently to implement Boolean logic gates. We begin by considering what is needed to construct an electronic computing device and generate corresponding constructs in GOL. Figure 7 shows the mappings we will use.

Electronic Computer	**GOL Computer**
Electrical pulses represent bits of data	Gliders act as bits of data
Wires transmit electrical pulses	Glider trajectories bring signals from one point to another
Processing devices associate inputs and generate Boolean results	Glider collisions occur, or do not occur, corresponding to signal inhibition and propagation (*false* and *true* results respectively)

Figure 7: Comparing an electronic computer to a GOL-based computer

Input Devices

With the electronic model of computing, an input signal will propagate only if the value is a logical *true* (logical high voltage). Since we are using gliders to represent data in the GOL computer, we must find a way to either generate or inhibit glider streams. To this end, we can represent a *false* input by combining a glider gun and a stopper. Of

course, to represent a *true* input we simply use a glider gun to generate a stream of gliders. These input constructs are shown in figure 8 below.

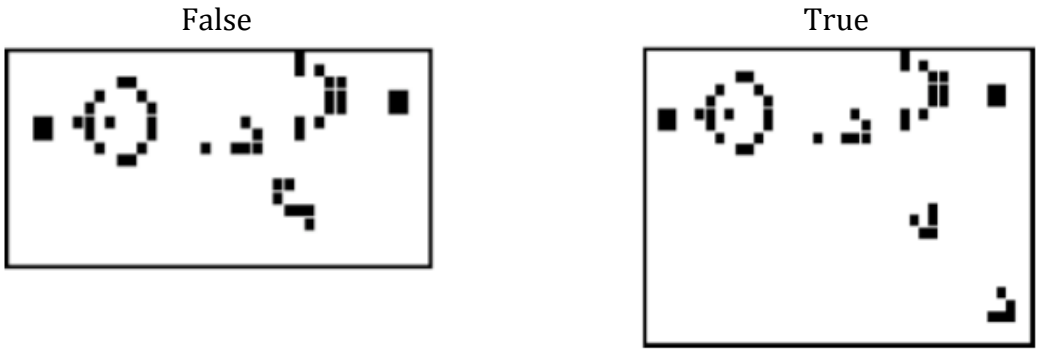

Figure 8: (Left) An inhibited glider stream representing false input. (Right) An uninhibited glider gun producing true input. (Rennard 2002)

Output Devices

The last thing that must be addressed before proceeding is the matter of output. What is needed here is a device that can be "activated" by a glider being emitted from a computation. That is, a device that is normally "off" but can be turned "on" by a glider collision. This device will be taken to represent the output of a Boolean function and when turned "on" will signal the end of computation. For our purposes, we will use a type of so called "eater" shown in figure 9.

Figure 9: (Left) an un-activated eater representing false output. (Right) An activated eater representing true output.

Now that we are able to fully represent binary inputs and outputs, we can consider how to form Boolean logic gates.

The NOT Gate

The NOT-gate is perhaps the most simple of the gates to construct. The device is comprised of two components, an input device and a glider gun. Figure 10 shows the arrangement of a NOT-gate.

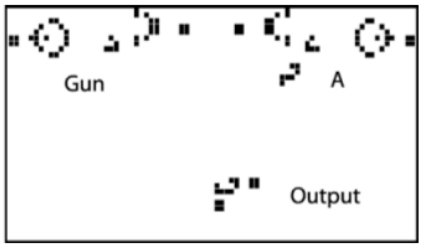

Figure 10: Implementation of Boolean NOT-gate in GOL. Here, A is false. (Rennard 2002)

The operation of this gate is relatively straightforward and is the basis for how all gates in this system work. Let us consider the two possible states of this gate:

Input A is false:

The input device on the right emits no gliders since input A was specified false. The glider gun on the left emits a glider stream that collides with the output device, activating it, and thereby indicating a logical *true* output.

Input A is true:

The input device on the right emits a stream of gliders since input A was specified true. The glider gun on the left emits a glider stream that collides with the stream being emitted by A. The glider streams annihilate one another and the output device is not activated, indicating a logical *false* output.

The AND-Gate

We now consider how to construct a Boolean AND gate. This gate will produce a *true* output if and only if both inputs are *true*. The construction for this gate is given in figure 11.

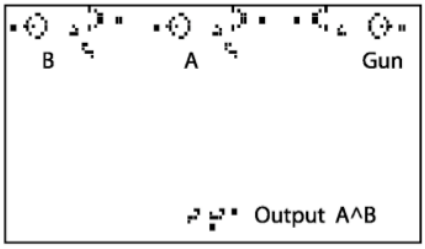

Figure 11: Implementation of Boolean AND gate in GOL. Here, inputs A and B are false. (Rennard 2002)

In this arrangement, the only glider trajectory in line with the output device (the output-enable stream) is that from input B. Also note that the trajectories of both Inputs A and B coincide with the trajectory of the glider gun at top right. So, we have four possible scenarios depending on the input values:

A and B are both false:

Neither A nor B emit gliders, the gliders from the glider gun terminate at the stopper next to the output device. The output device is never activated.

A is true, B is false:

Gliders emitted from A are annihilated by those from the glider gun. B does not emit gliders. Output device is never activated.

A is false, B is true:

A does not emit gliders. Gliders from input B are annihilated by those from the glider gun. Output device is never activated.

A is true, B is true:

A emits gliders that are annihilated by those from the glider gun. In essence, A blocks the glider gun from annihilating gliders being emitted by input B. Output device is then activated by gliders coming from B.

The OR-gate

Finally, we must provide a construction for the Boolean OR function. This gate will produce a *true* if either or both inputs are true. A construction for such a gate is given in figure 12.

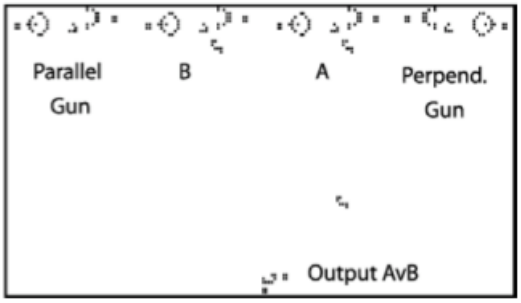

Figure 12: Implementation of Boolean OR gate in GOL. Here, inputs A and B are false (Rennard 2002)

In this configuration, the only glider stream that can activate the output (the output-enable stream) is that emitted by the gun at top left. This gun emits a glider stream parallel to inputs A and B. Note that the trajectories of the parallel gun and

inputs A and B all intersect the trajectory of the gun at top right. Once again, we have 4 possible cases to consider:

A and B are both false:

Neither A nor B emit gliders. The gliders from the perpendicular gun intersect and annihilate those from the parallel gun. Output device is never activated.

A is true, B is false:

B does not emit gliders. Gliders from A intersect and annihilate those from the perpendicular gun. Gliders from the parallel gun are able to reach the output device. Output device is activated

A is false, B is true:

A does not emit gliders. Gliders from B intersect and annihilate those from the perpendicular gun. Gliders from the parallel gun are able to reach the output device. Output device is activated

A is true, B is true:

Gliders from A intersect and annihilate those from the perpendicular gun. Gliders from B hit a stopper and do not interact with anything else in the gate. Gliders from the parallel gun are able to reach the output device. Output device is activated

Putting it all together

What remains to be said is how to connect the various logic gates we described above. It is not a particularly complex task, but one which requires that great attention be paid to how variables are grouped, as well as glider trajectories. Since we have at our disposal only 2-input logic gates, all Boolean equations must be grouped into expressions involving no more than two variables. For example, if we were to construct a logic circuit for A ^ B ^ C, we would begin by grouping (A ^ B) = A' and then ANDing this result with C. Of course, grouping is meaningful and this step must be performed in such a way that preserves original meaning. Certain techniques for parsing and grouping expressions in this way are studied mainly in compiler design and the details of these algorithms will not be explored here.

The next step is of course to link the various 2-input logic gates together. To do this, the output eater of the higher-level logic gate is removed, and the trajectory of the output-enable glider stream is oriented such that it aligns with the input of the lower-level gate. In the example above, we would need to remove the output eater from the

AND gate responsible for computing A', and place the gate in such a way that the output will appear as if it were an input in the lower-level gate that will then computer A' ^ C. Such an arrangement is shown in figure 13.

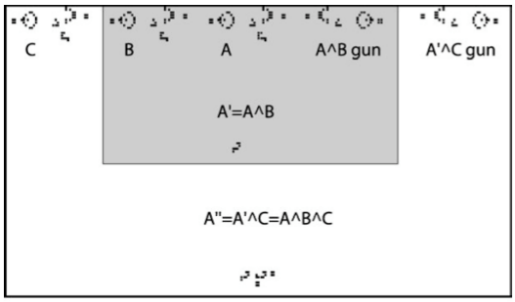

Figure 13: Nesting of logic gates to provide for complex Boolean functions (Rennard 2002)

Now that we are able to build simple logic gates and link them together to form complex Boolean functions, we have everything we need to build a fully functional Turing machine.

Further Exploration and Other Applications

While it is theoretically possible to build a Turing machine in the GOL environment, to actually carry this out is a monumental undertaking as the number of cells needed becomes very large. Furthermore, due to the highly-parallel nature of cellular automata, the amount of computing power necessary to efficiently run the machine, should it be built, is staggering. Regardless, some have built fully functional Turing machines within GOL. (see Rendell 2001)

The real power of cellular automata lie in their applications to mathematics, physical modeling, and algorithms. Stephen Wolfram has carried out a good deal of research in this area and has shown how cellular automata can be used for applications ranging from cryptography to fluid-dynamics. In his article *Cryptography with Cellular Automata*, Wolfram looks at the seemingly random behavior of certain cellular automata. He furthers this and examines how the pseudo-random behavior can be used for generating keys for data encryption. (Wolfram 1986)

Another fascinating application of cellular automata is in the theory of digital physics. The 1969 book *Calculating Space* by Konrad Zuse is credited as the first book on the subject. Zuse posits that the physical laws of the universe are discrete in nature and granting this, the state of the universe can be thought of as the state of a giant cellular automaton computing itself. The idea is often seen to be in conflict with other grand theories of physics which hold that physical laws are continuous, not discrete, and that not all occurrences in the universe happen deterministically. Regardless, there has been a fair amount of writing on Zuse's idea since the first publication of his book.

Cellular automata can also be leveraged as a useful tool to solve certain optimization problems. In a 2007 paper, researchers in Finland proposed an algorithm based on cellular automata for generating optimal solutions to forest-planning problems. The approach uses the grid of a two-dimensional cellular automaton as a map of the forest floor. Each cell takes on one of a number of states representing various characteristics of the trees on that plot of land. By encoding constraints into the rules of the cellular automaton, the algorithm produces a simulation of tree-growth in the forest. By running many simulations with varying characteristics, planting-plans that optimize, say, wood production can efficiently be found. (Heinonen & Pukkala 2007)

Summary and Conclusion

In this paper we have shown methods for constructing patterns in Conway's GOL cellular automaton that work as Boolean logic gates. With these constructions in hand, it is asserted that with careful planning, these components can be woven together to create a general purpose computing device with power equal to a Turing machine. We then looked at the draw backs of actually carrying out this implementation, and examined other areas in which cellular automata in general have been applied.

Attached to this paper is a Java-based implementation of a cellular automata simulator. The default rules for the simulator are those of Conway's GOL. The interface does however provide the user with the ability to customize the rules for both alive and

dead cells. Also available in the program is a rudimentary Boolean logic editor that allows users to create arbitrary logical statements and compute their result in the cellular automaton.

When implementing cellular automata simulators, the enemy is in the sheer number of individual calculations that have to be performed as the grid of cells gets larger and larger. As such, there are certain things that can be done to speed up the algorithms for a cellular automata simulator. The implementation provided here is relatively quick and dirty, but does do things such as only updating cells that the algorithms knows to be changing. This cuts down on the total number of calculations that have to be done each generation. Also, the quick lookup times provided by hashtables are leveraged by using them instead of lists to store the state of cells in the grid. Some particularly clever implementations utilize hashtables of known patterns and known evolutions in order to essentially skip ahead many generations at in a single time step when a "cached" pattern is recognized.

Works Cited

Wolfram, Stephen. Cellular Automata Los Alamos Science, 9 (Fall 1983) 2-21

Rennard, Jean-Philippe. "Implementation of logical functions in the Game of Life" Collision-Based Computing, Adamatzky A. (ed.), Springer, 2002.

Rendell, Paul. "Turing Universality of the Game of Life" Collision-Based Computing, Adamatzky A. (ed.), Springer, 2002.

Wolfram, Stephen. "Cryptography with Cellular Automata" Advances in Cryptology: Crypto '85 Proceedings, Lecture Notes in Computer Science, 218 (Springer-Verlag, 1986) 429-432

Heinonen, Tero and Timo Pukkala. "The use of cellular automaton approach in forest planning" NRC Research Press Web site at cjfr.nrc.ca on 20 November 2007.