Turing Completeness, Logic Gates, and the Universality of Conway's Game of Life

Candidate Number:

A Dissertation submitted to the Department of Mathematics
of the London School of Economics and Political Science
for the degree of Master of Science

2020/08/31

Summary

Mathematics includes deep questions into the very understanding of what it means to solve problems. Mathematicians of the early 20th century grappled with the very idea that not every problem could be solved. Now we know that not every problem has a computable solution but that we can build systems that can compute all solutions which do have a computable solution. These systems are called Turing complete and this concept lays the groundwork for modern computing. Achieving Turing completeness is not particularly difficult and this dissertation will focus on proving Turing completeness in the famouos Conway's Game of Life. This dissertation will explore how seemingly simple components can be created in Conway's Game of Life and arranged in such a manner that any possible computation could be carried out with a lattice of evolving cells.

Contents

List of Figures

Chapter 1

Introduction

Fundamental to mathematics is the notion of computability. The modern world relies on computational systems which all solve problems at incredible speeds but are bounded by both memory and time. These are not the only bounds on computation but there is also a bound on the problems that computers could even possibly solve given no restrictions. This notion of computability is an unprovable fact about the universe; no mathematical system can solve every problem algorithmically no matter how much time or space are available.

This limitation on computing solutions gives rise to the idea of a system that can compute every computable problem. These systems arise in many unexpected places and creating a system capable of universal computation is relatively easy to achieve. Proofs of universal computation exist for card games such as *Magic the Gathering*, print statements in the programming language C, and even the computer game *Minecraft*. Though each of these systems would make terrible computers in practice, given enough time and memory they are as powerful as any modern super computer given their ability to compute any computable function. Studying these fundamental notions of what it means to be a computer and the limits of computation go beyond silly examples of universality and have had a very important impact on how we approach problems.

This topic of Turing completeness and universal computation is a deep and fascinating topic with its roots in philosophy of mathematics, linguistics, and computer science. As a student of linguistics and of mathematics the connection between formal languages, human languages, computing has always been an interesting intersection between seemingly distant areas. Though this dissertation will not cover the particulars of human language and formal languages the ideas permiate throughout the topic. Topics such as recursive functions, lambda calculus, and binary trees, are just as much at home linguistics as they are in logic and mathematics.

Choosing to explore Conway's Game of Life gave me the opportunity to look at these deep ideas of computation within a visual yet abstract space. Conway's Game of Life is a well studied cellular automata allowing me to draw on a deep well of literature when exploring universal computation in a new space. Conway's Game of Life is also a very visual medium allowing for diagrams and simulations that aid in presentation and understanding of these complex topics. The nearly limitless expression of behaviours is a daunting feature of Conway's Game of Life and therefore the language of logic gates helps reel in the limitless possibilities to a space that is easier to manage and study.

This dissertation will explore the notion of Turing completeness within Conway's Game of Life, a 2-dimensional cellular automaton capable of complex behaviours. The dissertation will take a formal approach in showing that Conway's Game of Life is Turing complete by building up all the components needed to emultate the models of computation to achieve Turing completeness. Substantial work has been done in modeling Turing machines within Conway's game of life, these machines are never complete as it is impossible to model the infinite tape within such a simulation. These simulations are also extremely large in size and require substantial time to run even the simplest of examples. Rather than focus on this simulation aspect I will prove that completeness is possible using components built in Conway's Game of Life that allow for the buildup of advance logic and eventually leads to Turing completeness.

The next chapter will cover the basic notions of Turing Machines and Completeness as well as Cellular Automata and specifically the fundamentals of Conway's Game of Life. The chapter will also discuss models of computing and specifically how the Random Access Machine model is considered Turing equivalent. Chapter 3 will cover some of the previous work in the space including the earliest proofs of completeness as well as more recent work including proving the elementary cellular automaton Rule 110 is complete. Chapter 4 will cover the basic components required to build logic gates within Conway's Game of Life as well as the construction of the logic gates themselves. This will finally lead into the final proof showing that with these components the ability to use logic and thus the random access machine model can be implemented in Conway's Game of Life gives us Turing completeness.

Chapter 2

Preliminaries

This section will set out to explain the basic concepts needed for the dissertation. First we will look at the concept of a Turing Machine and Turing completeness. This leads into the key things that make a system Turing complete and the significance of this. Then this section will explain the specific system in question, namely cellular automata generally and more specifically Conway's Game of Life. Finally we will look at models of computation more generally showing that there are equivalent models of computation to Turing machines and the differing models of computation function together to build up more complex models of computing.

2.1 Computable Functions

In the early 20th century sparked a revolution in how we think about computing and what is computable. Three theories of computability were unified in the famous Church-Turing Thesis, which showed that different models of computing are equally powerful and gives the notion that is fundamental to studying if a function is calculable or not. This is a separate notion from the notion of computational complexity that asks about the informal notion of feasibility but rather the idea of whether a function can be calculated at all given no time constraints. Thus the idea of functionally calculable is incredibly powerful and lays much of the groundwork for modern digital computing.

2.1.1 Turing Machines

The notion of a computable function can be formalized into the language of Turing Machines, the hypothetical construct of mathematician Alan Turing. A Turing Machine is a theoretical machine that computes a fixed partial computable function,

which can be thought of as having a fixed program. Thus this is a tool that allows us to study computablity more easily. The machine consists of three main components: an infinitely long tape divided into cells, the read/write head, and the set of instructions. The values on the tape at the beginning is the input while values of the tape at the end are the output.

Formally a Turing Machine T as defined by Hopcroft and Ullman is $T = (Q, \Gamma, \Sigma, \delta, q_0, B, F)$ where:

- Q is the finite set of states of the head
- Γ is the finite set of tape symbols
- Σ is the finite set of input symbols and $\Sigma \subset \Gamma$
- δ is the transition function
- q_0 is the start state of the head where $q_0 \in Q$
- b is the blank symbol where $b \in \Gamma$
- F is the final accepting state and $F \subset Q$

At the start of the process all but a finite number of cells contain the blank symbol b. The cells are updated with the transition function which updates the state of the head and tells the head where to move next. The transition function $\delta(q, x)$ takes the current state of the head $q \in Q$ and the symbol in the current cell $x \in \Gamma$. The value of the function is defined as follows. $\delta(q, x) = (p, y, D)$ where:

- p is the next state of the head where $p \in Q$
- y is the next symbol that replaces x in the current cell where $y \in \Gamma$
- D is the direction in which the head moves, either L or R for left and right respectively

The transition function can be thought of as the instructions for the next step in the process. The functions tells us how to update the process depending on the internal state of the head and the symbol on the current bit of tape.

This relatively simple machine is as powerful as any system capable of universal computation. Thus it is a tool that allows us to study what is computable and incomputable. Based on the Church-Turing thesis if a function is computable then there exists a Turing machine that can compute such a function. Thus this simple system allows us to explore computability in Turing machine-theoretic terms. Building a single Turing machine is typically not enough to show completeness but there is a particular Turing machine that will get us closer to exploring complete systems.

2.1.2 Universal Turing Machines

While a Turing machine is capable of calculating a particular computable function a universal Turing machine is a turing machine that can simulate any Turing machine. Alan Turing describes a universal Turing machine U if supplied with the action table, or transition function, of a Turing machine T then U will compute the same output as T. Thus any computable problem, which is a problem a Turing machine can solve, will be computable by a universal Turing machine.

This concept has been formalized in the Universal Turing Machine Theorem which states,

Theorem 1. *Universal Turing Machine Theorem: For a universal Turing machine U there exists a computable function $f(i, x)$ where i is a string defining a particular Turing machine and x is an input string. Then for all i and x, $U(f(i, x)) = T_i(x)$ where T_i is a Turing machine.*

This theorem states that a universal Turing machine U given an input and a string representing Turing machine T will give the same output of Turing machine T given the same input. These machines are immensely more powerful since they are able to calculate any possible function rather than calculating a single particular function. These universal Turing machines are still based on the same mechanism as the classical Turing machine, and thus are immensely powerful when studying computing.

2.1.3 Turing Completeness

There are many ways to show a system is Turing complete but these methods all are essentially equivalent to proving the most basic definition,

Definition 2. A system is *Turing complete* if and only if it can solve every Turing Computable Function.

This is to say that since every Turing computable function can be computed using a Turing machine that a system capable of reproducing every Turing machine is Turing complete. By the universal Turing machine theorem there is such a function that can solve every Turing computable function and this function can be solved using a universal Turing machine giving us the following equivalent definition,

Definition 3. A system is *Turing complete* if it can simulate a universal Turing machine.

This definition is often employed when proving a system is complete, if it is possible to embed a universal Turing machine within a system then this is enough to show the system is complete.

2.2 Cellular Automata

Cellular Automata are discrete structures that consist of a grid of cells and states that each cell can be in. Each Cellular Automata has a set of rules for how the cells update creating an evolutionary behavior across the grid. The concept was invented by Stanisław Ulam and John von Neumann while exploring evolutionary systems and have proven useful in studying automata theory, biological systems, and physical systems.

2.2.1 Formulation

There has been a large amount of work done on cellular automata, each with its own flavour. Generally a Cellular Automaton can be formally described using four parameter, n-dimensional lattice of cells, finite set of states, neighborhood of interaction, and the set of rules for the evolution of the system. While extensions are possible and the variations within each parameter are vast these four parameters are typically the basis of a cellular automaton.

The first parameter, the *n-dimensional lattice* is broken up into discrete cells. These cells can be of any shape so long they fit into the lattice. The lattice is of finite dimensions but typically extends infinitely in any direction. The usual assumption is that each cell is identical and uniform throughout the lattice. For my research I will be focussing on square cells within a 2-dimensional lattice.

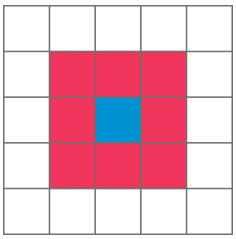
The second parameter *set of states* is a finite set Σ of finite size $|\Sigma| = k$ where each cell is in one state $\sigma_i \in \Sigma$ where $i \in 1, 2, \ldots, k$. The third parameter *neighborhood of interactions* relates to the state of neighboring cells. There are any number of possible neighborhoods, a common neighborhood is the Moore Neighborhood.

Definition 4. The *Moore Neighborhood* in a 2-D lattice for a given cell at position (x, y) are the 8 cells with position $(x \pm 1, y \pm 1)$, $(x, y \pm 1)$, and $(x \pm 1, y)$

In this neighborhood the state of a cell is affected by the state of the 8 neighboring cells. Figure 2.1 shows the Moore Neighborhood of the blue cell in the center are the red cells immediately surrounding it.

Finally the Cellular Automaton needs a *set of rules* that govern the evolution of the system. This can be thought of as a map $\Sigma^n \to \Sigma$ where Σ^n is the states of

Figure 2.1: Moore Neighborhood



the n neighbors that determine the new state $\sigma \in \Sigma$ of the cell being updated. The state of a cell at time t is dependent on the states of the neighbors at time $t-1$. Typically the update is synchronous, meaning the cells update simultaneously, this type of updating will be used in the rest of the dissertation.

2.2.2 Conway's Game of Life

Conway's Game of Life is a particular Cellular Automaton invented by John Conway and popularised by *Scientific American* writer Martin Gardner in the magazine's October 1970 issue. Conway's Game of Life is a 2-dimensional, 2-state cellular automaton that operates on the Moore Neighborhood. The Automaton was named the "Game of Life" as it simulates the birth and death of living organisms. The game's two states are often referred to as *alive* and *dead*, which can also be referred to as 1 and 0 respectively.

The rules of the game can be summarized by three rules pertaining to the living cells and are as follows:

- Survival: A cell will remain alive if only two or three neighboring cells are alive
- Birth: A cell is born, or changes from dead to alive, if exactly three neighbors are alive.
- Death: A living cell dies if less than two neighbors or more than three neighbors are alive.

These rules give rise to chaotic behaviours as well as complex non-chaotic behaviour and yields itself to a deep study in the structures that arise within the game.

There are a few key ideas in Conway's Game of Life that are important to know when studying. The grid, sometimes referred to as the *universe*, updates simultaneously and each step forward in time is referred to as a *generation*.

Within the Game of Life there are patterns, sometimes referred to as objects, which make up the landscape.

Definition 5. A *pattern* is any particular configuration of cells in the infinite universe.

This notion is less useful for studying the particular sections or interations within the universe. By establishing a bounding box around a particular area we can study finite patterns that do not encompass the entire universe.

Definition 6. A *finite pattern* is a particular configuration of cells within a bounding box where all cells outside the bounding box are dead.

This idea of a finite pattern allows small objects with their own unique properties to be studied. A finite pattern may move across the universe or produce other finite patterns. Finite patterns include non-changing patterns, referred to as a still life, such as the block in 2.2 or changing patterns such as oscilators such as the blinker shown in 2.3.

Figure 2.2: Block

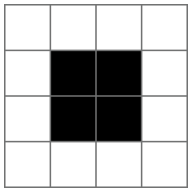
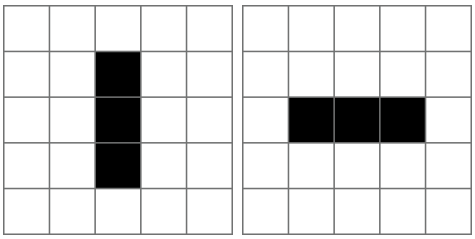


Figure 2.3: Blinker



The block has a bounding box of 4×4 cells. Between generations there is no change and therefore the bounding box includes the living cells neighborhood around each living cell only as those are the only cells that affect this finite pattern. The blinker has a bounding box of 5×5 cells. Similar to the block the bounding box includes the neighborhoods of all living cells, but since this finite pattern oscillates between the vertical and horizontal states as shown in 2.3 the bounding box will need to accommodate both states. This means it will be larger than it appears at any one generation.

By placing multiple finite patterns within the same universe, interactions, referred to as collisions, between these finite patterns become possible.

Definition 7. A *collision* between finite patterns is the interaction when the neighborhoods of cells within two separate finite patterns overlap.

Collisions are incredibly important to complex behaviours within Conway's Game of Life as they can lead to both the destruction of a finite pattern or give rise to new patterns. Certain finite patterns exists that will remain intact after a collision. Not all collisions will be able to have a recovery, and most finite patterns that can recover may only recover after a particular collision on a particular part of the finite pattern. These interactions may destroy or create new finite patterns as with normal collisions but they may recover and be used again after a number of generations known as the recovery time.

Definition 8. The *recovery time* of finite pattern is the number of generations after a collision until the finite pattern returns to its original state.

These finite patterns with the ability to recover after a collision are useful as they may be reused after a collision. An example of finite patterns that can recover is the class of pattern known as eaters. The simplest eater is the block shown in figure 2.2 which can recover to its original state after certain collisions.

Finite patterns also exhibit an important property called heat. A finite pattern's *heat* is the average number of cells that change state per generation. A blinker's heat, with its two state shown in 2.3, is four as two cells die and two cells are born per generation. The heat of a still life will always be zero since no cell changes state between generations.

The information transfer in the Game of Life is bounded by the so called *speed of light* c of Conway's Game of Life

Definition 9. The *Speed of Light*, c, is a displacement of one cell per generation.

The speed of an object is how fast an object travels across the universe and is always defined in terms of c

Definition 10. The *Speed*, v, of an object is defined as follows:

$$v = \frac{max(|x|, |y|)}{n}c$$

- n is the period of the object
- $|x|$ is the horizontal displacement
- $|y|$ is the vertical displacement
- c is the speed of light

The notions of speed and speed of light can be modified to work with other cellular automata and the study of these properties extend much further than Conway's Game of Life.

2.3 Logic Gates

Logic gates are gadgets that can perform logical operations. Logic is fundamental to mathematics and computation and will play a key part in the final proof of completeness. This section will go over the basics of logic gates and the notion of functional completeness.

Definition 11. A set S of logical connectors is *functionally complete* if all possible truth tables can be expressed by members of the set S.

Functional completeness tells us that all possible logic gates can be constructed from the set. To achieve functional completeness through logic gates we will need to construct enough gates to represent all possible truth tables.

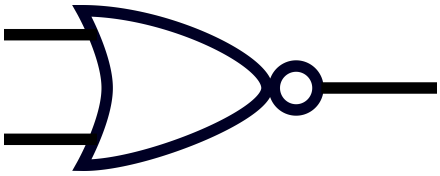
Definition 12. A *Logic Gate* is a device which performs a logical operation on one or more binary inputs and has one binary output.

The set of logical connectives $\{\wedge, \neg, \vee\}$ is functionally complete, meaning with these three gates any possible truth table can be constructed. But there are smaller sets that are also functionally complete. The set that this section will focus on is the set $\{\downarrow\}$ or the set containing logical NOR. The NOR gate, shown symbolically in figure 2.4, is used often in modern computer architecture, many of the simpler gates within modern specifications require an inverter, an end negation, making the gate more complex. The NOR gate's output is 0 unless both inputs are 1 as seen in 2.1. Similarly within Conway's Game of Life there is a simple method for constructing a NOR gate which will be discussed later.

Table 2.1: Boolean Logic Values

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \downarrow q$
1	1	0	1	0	0
1	0	0	0	1	0
0	1	1	0	1	0
0	0	1	0	1	1

Figure 2.4: NOR Gate



Before constructing circuits using the NOR gate proving the functional completeness is important. The following theorem and proof rely on the fact that the set $\{\neg, \wedge, \vee\}$ is functionally complete.

Theorem 13. *The singleton set of the logical connective* $\{\downarrow\}$ *is functionally complete.*

Proof. Given statements p and q
$\neg p \equiv p \downarrow p$

Given statements p and q
$p \wedge q \equiv (p \downarrow p) \downarrow (q \downarrow q)$

Given statements p and q
$p \vee q \equiv (p \downarrow q) \downarrow (p \downarrow q)$

Thus, $\{\downarrow\}$ is functionally complete. □

2.4 Circuits

An aside to Turing completeness and Conway's Game of Life that will become important in this dissertation's later constructions is the notion of mathematical circuits. There are two main types of circuits that will be needed when exploring completeness in chapter 4. The first type of circuit is a combinational logic circuit and the second is a sequential circuit.

Definition 14. A *combinational logic circuit* C is defined by the following triplet (G, E, L) where

- G is a labeled acyclic graph
- E is the set of edge values $\{0, 1\}$
- L is the labels of verticies corresponding to the logic gates each of which is a function that takes some values from E^i to E where i is the in degree of the gate.

Each vertex is labeled with $l \in L$ representing the logic gates and each edge is labeled from $e \in E$ representing the values of the information being carried.

In less formal terms the edges of the graph represents the wires carrying data and the vertices represent the logic gates. These circuits' output is only dependent on the input of the circuit and therefore has no memory when performing tasks. These circuits are capable of Boolean algebra for data transformations so long as the set of logic gates labels L represent a set of functionally complete logic gates.

A combinational logic circuit is never capable of being a Turing complete system as it lack memory or recursion. This is where the next type of circuit will come in, the sequential logic circuit. This is a circuit that allows cycles and is defined as follows.

Definition 15. A *sequential logic circuit* C is defined by the following triplet (G, E, L) where

- G is a labeled graph
- E is the set of edge values $\{0, 1\}$
- L is the labels of verticies corresponding to the logic gates each of which is a function that takes some values from E^i to E where i is the in degree of the gate.

Each vertex is labeled with $l \in L$ representing the logic gates and each edge is labeled from $e \in E$ representing the values of the information being carried. Input elements are $l \in L$ with indegree 0 and output elements are $l \in L$ with outdegree 0.

The sequential logic circuit as described above allows for circuits that can depend on previous states of the circuit. This allows for memory within the system as future parts of the circuit can rely on previous outputs. This gives rise to an equivalent definition from Savage.

Definition 16. A *sequential logic circuit* is a loop of combinational logic circuits and storage elements with an input and output.

These circuits are more powerful than combinational logic circuits and allow for more sophisticated calculations. These circuits will be important in the final proof of Turing completeness in Conway's Game of Life but the next section will discuss the next important step namely finite state machines and register machines.

2.5 Register Machines

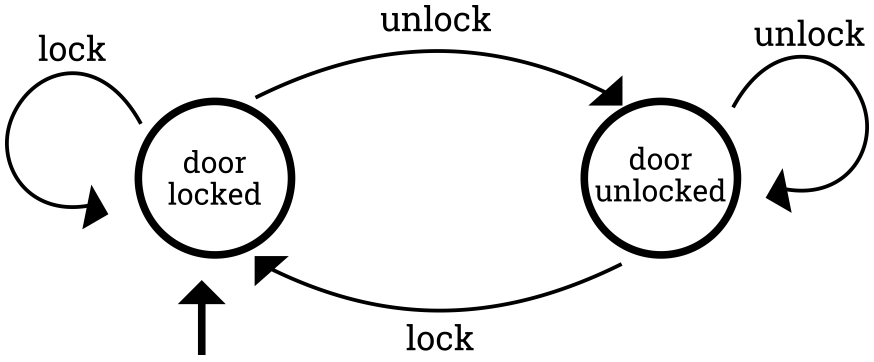
There are many models of computation that exists alongside the Turing model. These models have differing speeds and sometimes differing capabilities. One such model is the Finite State Machine as defined by Savage

Definition 17. A *Finite State Machine* is defined by the following tuple $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$, where

- Σ is the finite set of input symbols
- Ψ is the finite set of output symbols
- Q is the finite set of states
- $\delta : Q \times \Sigma \to Q$ is the next-state function
- $\lambda : Q \to \Psi$ is the output function
- s is the initial state
- F is the final accepting state and $F \subset Q$

An illustration of a finite state machine can be seen in the following state diagram for approaching a door. This diagram shows the states, Q, where the door is either open or closed. The vertical arrow represents the initial state s where the door is closed and the transition functions are the edges between the verticies. This simple example illustrates how finite state machines opperate under a finite set of states and transition from one to another.

Figure 2.5: Finite State Machine Example



A finite state machine relies on moving to a different state depending on the internal state and the next state functions δ. This can be simulated by a sequential circuit [14]. The following proof adapted from Savage.

Theorem 18. *A Finite State Machine can be simulated by a sequential circuit.*

Proof. For a given finite state machine M and a sequential logic circuit C the states Q can be translated to binary by a functions $f_Q : Q \to B$ and are stored within the storage elements in C. The input symbols Σ can be translated to binary with a function $f_\Sigma : \Sigma \to B$, with initial state s being the initial state of the circuit. Finally the input symbols Ψ can be translated to binary with a function $f_\Psi : \Psi \to B$.

The transition functions δ can be translated into combinational logic circuits that transition from one storage element to the next element representing the next state. Since each state is stored as binary the state needs to take $\delta : B \to B$. Similarly the output function takes a binary input to a binary output. $\lambda : B \to B$. This can be represented by combinational logic circuits which take a binary input to a binary output. □

A finite state machine is not as computationally powerful as a Turing machine but the head of a Turing machine can be considered a finite state machine [14].

Theorem 19. *The head of a Turing machine is a finite state machine*

Proof. Given a finite state machine M and Turing machine T. The sets input and output symbols Σ and Ψ in M are equivalent to the sets tape symbols being read and written from the head in T. The initial state s drawn from a set of finite states Q in M is identical to the initial state of the head of T. The transition function δ is the same in M and T. The final accepting state of M reached when the halting symbol b from the tape T.

Thus the head of a Turing machine is a finite state machine. □

The finite state machine does not have the same computational power as a Turing machine, but this model of computation is a useful tool when describing more powerful computational model. Beyond the finite state machine there exists a class of models that are all considered equally computationally strong as Turing machines and these models are called Turing equivalent.

Definition 20. A system P is *Turing equivalent* if the P can be simulated by a Turing machine and a Turing machine can simiulate P.

Many different Turing equivalent systems were developed throughout the 20th and 21st centuries with one of the most studied class of Turing equivalent machines being register machines. These machines rely on registers that can be accessed as needed. One such register machine is the random access machine as defined by Cook and Reckhow[4].

Definition 21. A *Random Access Machine* (RAM) is a finite state machine operating on an infinite amount of registers.

This construction uses a finite state machine as the control mechanism in a similar fashion to the Turing machine. The inputs come from the addressed registers $X_0, X_1, X_2, \ldots$ that can each hold an integer value. A major benefit to the RAM as opposed to a Turing machine is the ability to look up memory locations randomly rather than only able to move one memory cell at a time. The states and state functions of the RAM are typically described as a table of instructions which are read sequentially unless a jump occurs.

Definition 22. A *jump* is a departure from the sequence of instructions to another sequence

While the table of instructions can differ, a simple example that preserves the power of the random access machine is described by Minsky and includes the following instruction set:

- Increment
- Decrement
- Jump if Zero
- Halt

[8]

These instructions take a value from the indicated register and performs the instruction register before moving to the next instruction. The state register of the machine is read and depending on the state the next instruction is executed. There are typically five types of instructions associated with a random access machines: logical instructions, read and write instructions, jump instructions, input-output instructions, and the halt instruction. The instructionis are stored as a finite state machine with certain registers being used for storing the state and the current values needed for calculations. The following theorem and proof is adapted from Savage and shows that a random access machine is Turing equivalent[14].

Theorem 23. *A Random Access Machine (RAM) is Turing equivalent.*

Proof. This proof requires proving two directions that an RAM can simulate a Turing machine and a Turing machine can simulate a RAM.
First we will prove that RAM can simulate a Turing machine. Since the head of a Turing machine is a finite state machine and the control of a RAM is a finite state machine it suffices to show the RAM program can simulate a Turing machine tape.

This can be done by storing an extra word stored in the RAM register. This word is incremented and decremented and points to the next register to read simulating the right and left movement of the Turing machine head along the tape. Thus simulating a Turing machine tape.

Now we will show that a Turing machine can simulate RAM. Take a multitape Turing machine. The first tape stores pairs (i_j, c_j) with a symbol marking the boundary. Then for each register in RAM j the contents is represented by c_j and indexed by the Turing machine by i_j. The second tape is equivalent to the accumulator, registers used for calculations, and a third tape for scratch work. The final tape is used for the input and output of the RAM. The Turing machine executes the instructions using the head as a finite state machine. The tape can be searched by moving the head right until symbol i_j is found. The contents c_j is moved to tape 3, and any calculations can be done with the contents of tape 2. When the halting symbol is found the output is copied to tape 4. Thus a Turing machine can simulate a RAM.

Since a RAM can simulate a Turing machine and a Turing machine can simulate a RAM the computational models are equivalent and thus RAM is Turing equivalent. □

Chapter 3

Previous Work

This section will cover some of the research in Turing completeness and Cellular Automata. Research in Automata theory has been vast with applications in physics and biology as well as in computing. Aspects of this research has been used in proving these discrete systems with relatively simple rules could be capable of achieving turing completeness. Work beginning in the 1970s helped pave the way for proving universal computation in cellular automata. A proof embedding Turing Machines into 1-dimensional cellular automata laid the groundwork for future proofs that gave specific examples of complete automata. It was not until 2008 that a proof emerged that a particular 1-dimensional cellular automata was Turing complete.

3.1 General Completeness with UTM

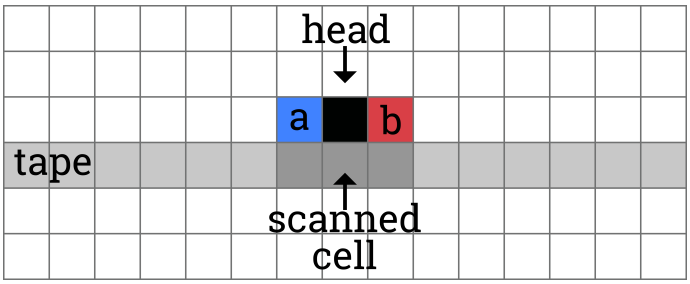
Turing completeness of cellular automata is a topic that has been studied since the 1970s. Early studies showed that 1-dimensional and 2-dimensional cellular automata have the necessary ingredients for completeness. Alvy Ray Smith, the computer scientist known for co-founding Pixar, showed that it is theoretically possible to embed a universal Turing machine into a cellular automata in one and two dimensions. But it was not until 2008 where a particular 1-dimensional cellular automaton was proven to be Turing complete.

Smith's paper lays out two general methods for proving Turing completeness of cellular automata. The first method involves embedding a universal Turing machine within the system. Leading to the first major theorem in the paper.

Theorem 24. *For an abritrary Turing machine T with m tape symbols and n states, there exists a 2-D, 6-neighborhood, $max(m+1, n+1)$-state cellular automata Z_T which simulates it in real time.*

The following example depicts a 2-D, 6-neighborhood cellular automaton. The number of states required for the construction is the larger value of either states or tape symbols of the Turing machine that is being emulated. Figure 3.1 depicts the construction described in 24. The gray cells depict the infinite tape in the construction. The black cell labeled *head* acts as the head of the Turing mahcine and moves across the tape aided by cells a and b which act as controls helping to move the head by being the only cells that can transition to the head state in the next generation. The cells directly to the left and right of the scanned cell shown in darker gray are the only cells besides the head, scanned cell, and a and b cells that change after a generation. This construction given the correct number of states and the correct transition functions can simulate a Turing machine as the head cell moves along the tape cells.

Figure 3.1: Theorem 24 Construction



This construction will not allow for a direct proof of the completeness of the Game of Life. Smith describes a 6-state 6-symbol universal Turing machine proposed by Minsky as Turing machine that proves universality of this cellular automata. Wolfram has described a universal 2-state 5-symbol Turing machine and conjectured the existance of a 2-state 3-symbol universal Turing machine [17]. These Turing machines would at best require a 3-state cellular automaton in this construction while Conway's Game of Life is a 2-state automata. Minsky argued that 2-state 2-symbol universal Turing machines could not exist [8]. Thus this strategy cannot prove the completeness of Conway's Game of Life this early proof laid the groundwork for proving completeness in 2-dimensional cellular automata.

Smith also proves a theorem for a similar construction in the 1-dimensional case.

Theorem 25. *For an abritrary Turing machine T with m states and n tape symbols, there exists a 1-D, 6-neighbor, max $(m+1, n+1)$-state cellular automata Z_T which simulates it in 3 times real time.*

A similar proof in the 1-dimensional space was also proven. This construction took the construction from theorem 24 and modified it to function in one dimension

by allowing the neighborhood of a cell to extend three cells in either direction. This construction means the head must move 3 cells to simulate the next step and scan the next cell equivalent to the tape. Thus this constrution functions in 3 times real time. This construction would be possible within 2-dimensions but is slower than the previous construction.

3.2 Post Tag System and Rule 110

Alvy Ray Smith also lays out a method of proving Completeness by using a Post tag system.

Definition 26. A *Post tag system* is defined by (m, A, P):

- m an integer called the deletion number
- A a finite alphabet
- *P(x)* a set of production rules where $x \in A$
- H a halting symbol $H \in A$

The Post tag system works given a word S deleting m letters on the left side of a word and appending $P(x)$ where $x \in A$ is the leftmost letter of S. This continues until the halting symbol H appears in the leftmost position. This was developed by Emil Post in the 1940s as a way of studyinig decision problems, though the Turing universality was explicitly not proven in this paper [10]. Minsky proved the Turing equivalence of the Post-Tag System in 1961 by simulating a universal Turing machine with the tag system thus giving another method for studying Turing completeness.

Smith uses this tag system proving the theorem 27 based on the m, the deletion number, in the Post-Tag system. This is done by changing the states in the leftmost cells where each state of the cell represents a letter $x \in A$ and the state changes are representations of the production rules, $P(x)$, in the tag system. If the halting symbol H appears in the left most position the whole process stops.

Theorem 27. *For an abritrary Post tag system T, there exists a 1-D 2-neighbor cellular automata Z_T which simulates T in $(m + 1)$ times real time.*

This tool has been used in the study of computability theory and gained more notoriety for its use in studying elementary cellular automta. This system is simple and allows for easy translation into simple systems such as Rule 110 cellular automata.

Rule 110 is a particular elementary cellular automaton which as of August 2020 is the simplest cellular automata to be proven Turing complete. Rule 110 is a 1-dimensional 2-state cellular automaton. Defined as follows

Definition 28. *Rule 110* is a 1-dimensional cellular automata with two states 0 and 1 and a transition function as follows:
For a cell C_i in the universe of the automaton the transition function for this cell $f(C_{i-1}, C_i, C_{i+1})$ is definied as follow:

- $f(0,0,0) = 0$
- $f(0,0,1) = 1$
- $f(0,1,0) = 1$
- $f(0,1,1) = 1$
- $f(1,0,0) = 0$
- $f(1,0,1) = 1$
- $f(1,1,0) = 1$
- $f(1,1,1) = 0$

Rule 110 has been characterized as an automaton where complex local behaviour may arise and is not purely stable or chaotic[17]. Conjectured by Stephen Wolfram in 1985, the completeness of Rule 110 was finally proven in 2008 by Matthew Cook. The proof used by Cook used a cyclic tag system developed to show completeness in the paper.The cyclic tag system described by Cook was created to bypass the problem of needing a random access lookup table, while maintaining the tag system's Turing equivalence.

The cyclic tag system is a modificatiton of the Post tag system that works in the following way. A word S is read from the left and appended on the right as in the Post tag system. Instead of using an alphabet A and a set of production rules $P(x)$ where $x \in A$ the items that are appended to the end of S are defined by a list that cycles back to the beginning when the end of the list is reached. The cyclic tag system consists of only two symbols 0 and 1 and these synmbols determine if the next item in the list is appended or not. If 1 is read from S then the item on the list is appended to the end. If 0 is read then the item is skipped. In both cases the leading letter is deleted.

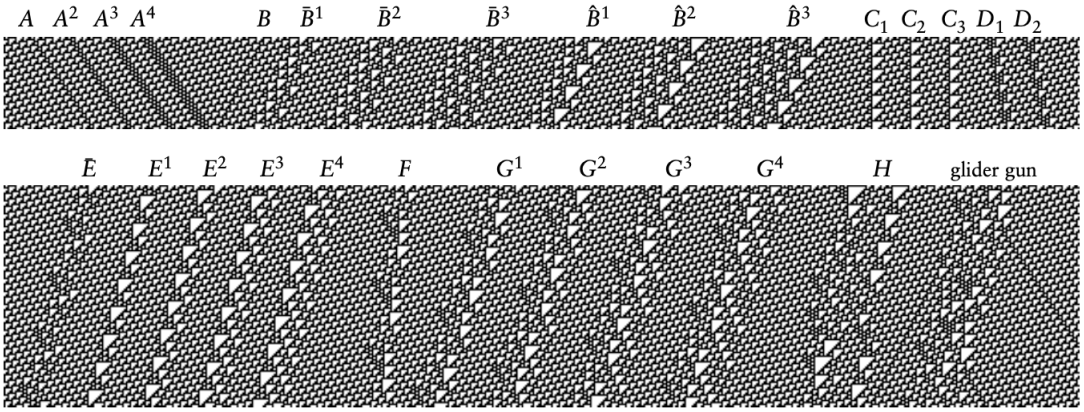
Given the example word `1010` and a list `(01,11,00)` the following example illustrates a simple cyclic tag system procedure:

```
Example Procedure:
list      action     word
```

10	append	1010
11	skip	01001
00	append	1001
10	skip	00100
11	skip	0100
00	append	100
10	skip	0000
...	...	...

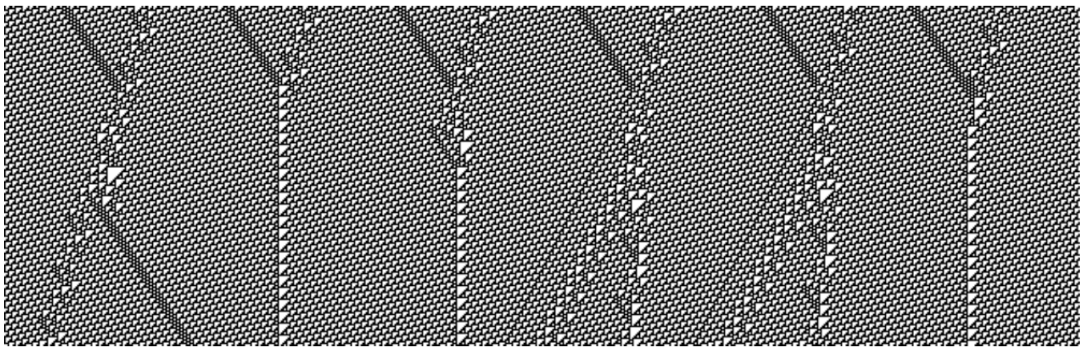
Once there are only zeros left in the word the process will eventually halt as nothing will be appended to the end. The case presented halts but in many cases the process will never halt as the word will continue to be appended forever. The cyclic tag system is universal as shown by Cook as it can emulate a Post tag system thus gives a tool for working within the space of elementary cellular automata.

Figure 3.2: Gliders in Rule 110 [3]



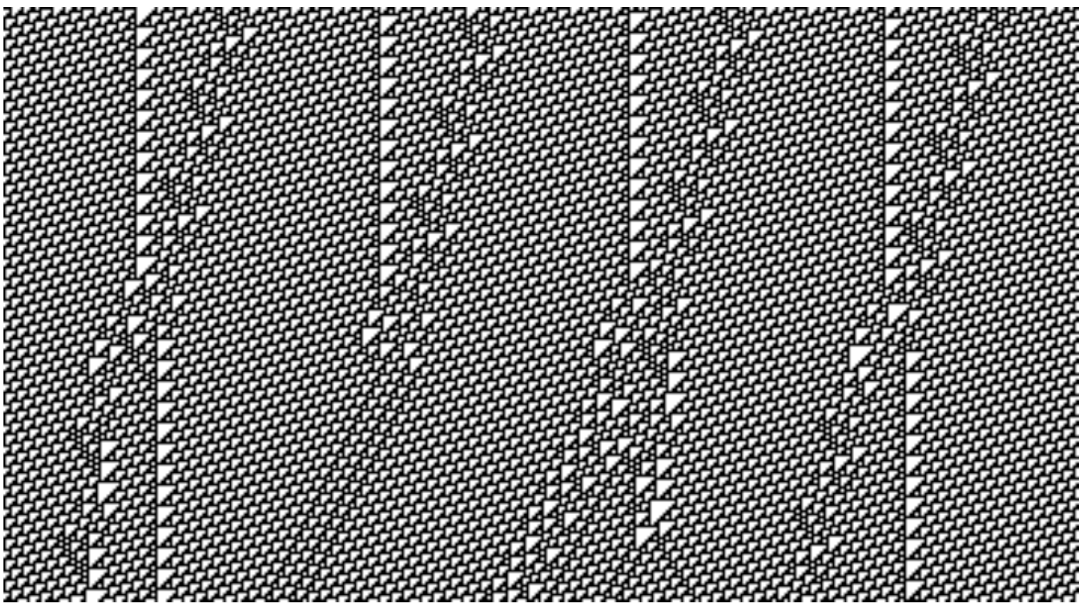
To emulate the cyclic tag system in Rule 110 Cook developed a system of information transfer using a finite pattern called a glider. Gliders in Rule 110 are finite patterns that move across the universe of the automaton eventually returning to their original configuration shown in figure 3.2. Gliders in Conway's Game of Life are similar and will be discussed later. Several types of gliders in Rule 110 exist and it is the collisions between two gliders that give the power to emulate the cyclic tag system. While the setup is quite complex the general idea is to use three gliders, the $\bar{E}$ glider, the C_2 glider, and A^4 glider. Given a cyclic tag system with word S and a list of symbols to be appended, the system can be represented by the three gliders. The C_2 gliders are used to represent sybols in the word in S and the $\bar{E}$ along with the A^2 gliders are used to append to symbols to S.

Figure 3.3: Collisions Between $\bar{E}$ glider and A^4 glider [3]



The system functions by having the word S from the cyclic tag system represented by vertical gliders. A vertical glider, such as the C_2, glider does not move left or right across the universe and rather oscillates in place. Gliders representing the list from the cyclic tag system come in from the right and when they collide with the symbol representing 0 the collision eliminates the incoming symbol along with the leading symbol, 0. This process emulates the list item being skipped when hitting a 0. When the incoming glider representing the next list entry collides with a symbol representing 1 it eliminates the leading 1 symbols and passes through all the symbols behind the leading symbol and gets appended to the end as a vertical glider representing the word to be appended.

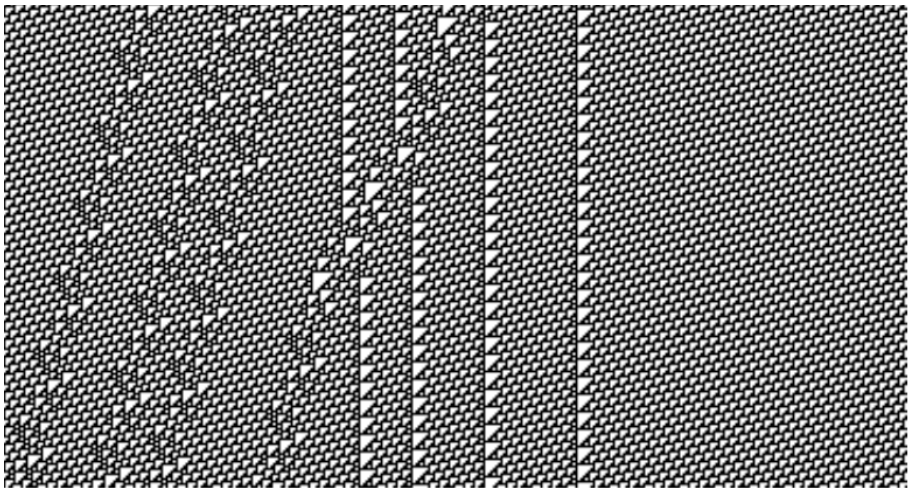
Figure 3.4: Collisions Between $\bar{E}$ glider and C_2 glider [3]



The particular construction of each glider governs their interaction with one another. The $\bar{E}$ gliders and C_2 gliders can cross over one another without destroying

the pattern shown in the rightmost collision in figure 3.4. Figure 3.3 shows the 6 possible collisions between the $\bar{E}$ glider and the A^4 glider with each row of cells representing the next generation forward in time. Several possible collisions between A^4 glider and $\bar{E}$ glider creates a C_2 glider. In the particular construction the glider itself does not represent the symbols in S directly, but rather the spacing between two C_2 gliders represents the bit of data 0 or 1. A narrow gap being a 0 and a wider gap being a 1 is shown in figure 3.5 by the vertcal lines of C_2 gliders.

Figure 3.5: Spacing between C_2 Gliders[3]



Since the $\bar{E}$ glider can pass through the C_2 glider and a $\bar{E}$ glider collision with a A^4 glider produces a C_2 glider these are the perfect candidates for creating the cyclic tag system. The word S is represented by a series of C_2 gliders and the incoming $\bar{E}$ gliders representing the list of words can either be rejected or allowed by the leading symbol. If accepted it moves through the C_2 gliders and is appended by a A^4 glider collision at the end of the word S. Depending on the spacing of the collision this in turn represents a 0 or 1 being appended. This process continues, or eventually halts once there are only 0 represented in the word. Through this complex construction with the correct assembly Rule 110 can emulate a cyclic tag system which emulates the Turing equivalent Post tag system and thus proves that Rule 110 is capable of universal computation.

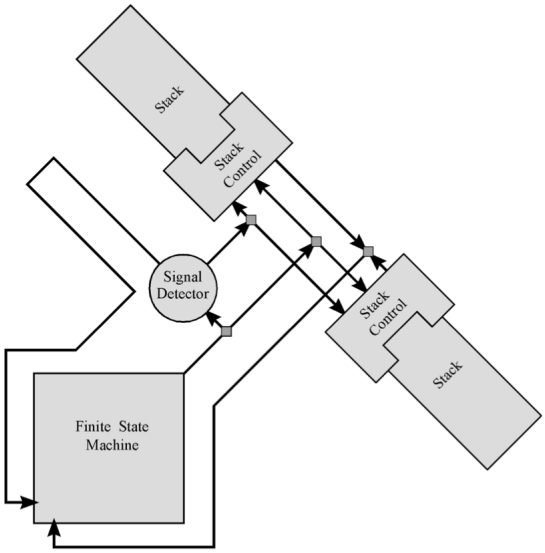
3.3 A Universal Turing Machine in the Game of Life

John H. Conway is said to have began work on proving the Turing completeness of Conway's Game of life based on register machines in the 1970s [17]. This long

standing belief in Turing completeness of Conway's Game of Life was eventually shown to be true through implementations of universal Turing machines within Conway's Game of Life. These implementations were made possible by the non-chaotic and complex behaviours exhibited in Conway's Game of Life, leading to Wolfram classifying it as a class 4 cellular automata. This class includes Rule 110 as well as Conway's Game of Life and it is conjectured that many automata this class are capable of universal computation.

Paul Rendell much like Alvy Ray Smith III designed a proof of Turing completeness by creating a Turing machine within a cellular automata. Unlike the smaller Turing machine that Smith created this Turing machine required many extra cells to cope with the state and rule constraints presented by Conway's Game of Life. With only two states and few rules the complex interactions that appear can substitute for lack of states and transitions with complex structures created to store multiple states and internal logic to govern the transitions. This construction relies on a *glider* which is a finite pattern that travels across the universe. More detail will be given on this finite pattern in the next chapter.

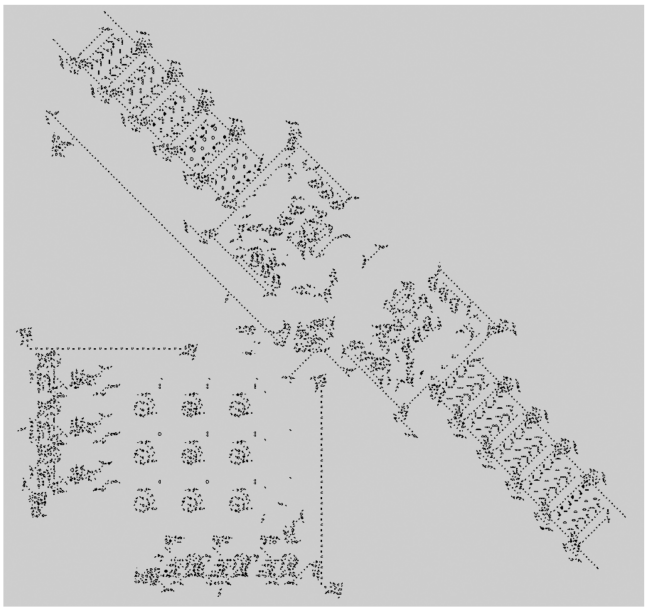
Figure 3.6: Rendell's Turing Machine Design [12]



This Turing machine opperates on stacks of *memory cells* which trap gliders, the information transmission medium, within the cell and a finite state machine to act as the head as seen in figure 3.6. Given the infinite size of the grid in the Game of Life Rendell used two stacks of memory cells that represented the infinite tape as seen in figure 3.7. The stacks were postitioned at either side of the head and as one pushed the other stack popped in a sense moving the tape past the head as needed.

A signal detector readied the head and the stack for the read-write operation and the head would perform the correct operations before writing back to the stack of memory cells.

Figure 3.7: Rendell's Turing Machine in Golly [12]



The universal Turing machine has 13-states and 8-symbols. The memory cells in the stack utilize three gliders to store the symbol, with the symbols stored in binary with a glider representing 1 and lack of glider representing 0. The finite state machine includes gliders to represent the internal state, the read symbol, and the next position to move on the tape. The Turing machine halts if all the glider streams are missing. This machine requires 11040 generations per cycle, but even with the large amount of generations and large size this definitively proves the completeness of Conway's Game of Life [12].

3.4 Hashlife and Simulations

Simulating any cellular automata can be done with a pen and paper. Writing out the states by hand one generation at a time can allow for the study of these objects to be undertaken. In reality this method is much too slow to gain meaningful insights into automata since most of the small to structures have been constructed already. This is where computing comes in to help show more sophisticated structures and analyze much larger objects and interactions. Writing a program to simulate the Game of Life is relatively simple. With three update rules and two states these rules are easy to implement with conditionals checking each cell for an update.

This formulation is quite inefficient when working on larger complex structures. Much like the pen and paper method a more efficient computational method is required for quicker and more complicated simulation. The need for a faster computational method lead to Bill Gosper's invention of the `hashlife` algorithm that uses quadtrees and hashing to speed up the computing of the states. The actual implementation of hashlife is quite complex but there exists an open sourced implementation called `Golly` created by Andrew Trevorrow and Tomas Rokicki. `Golly` is widely used as it allows for graphical look at Conway's Game of Life using the hashlife algorithm.

This is the software that was used in the construction of the Turing machine by Paul Rendell. The ability to script components, share code, and run simulations quickly made the ability to create this simulation possible. Beyond construction of the universal Turing machine this ability to computationally study Conway's Game of Life has lead to the discovery and creation of many new components whether it was observing patterns that emerge from random starting states or the ability to code sophisticated scripts that build complex structures within `Golly`. Most current studies of Conway's Game of Life have benefitted from the development of the `hashlife` algorithms and programs such as `Golly` that implement it. This dissertation relies on small simulations run in `Golly` but unlike Rendell's proof, an entire Turing machine will not be constructed using `Golly`.

Chapter 4

Completeness in Conway's Game of Life

Turing completeness has been shown in a variety of Cellular Automta, including Conway's Game of Life. This section will focus on my own formulation for proving Turing completeness using logic gates. Though logic gates have been built in Conway's Game of Life this proof will focus on the components to build circuitry and how these components simulate the correct models of computation. Lanes of gliders, with more details below, will act as our information transfer medium. A lane containing gliders will act as our *on* or 1 state while the lack of the gliders will be considered *off* or 0. This use of this information transfer will diverge from some of the previous proofs of completeness, focussing on the ability to build up components that allow for the construction of logic gates, sequential circuits, and eventually a random access machines thus giving us a Turing complete system.

4.1 Building Blocks in Conway's Game of Life

The ability to create structures within Conway's Game of Life that are non-chaotic and have periodic behaviours act as the catalyst for building logic gates, and eventually proving completeness. These elements are well studied and new interesting objects and structures are being discovered all the time. Many of the simplest elements were discovered during experimentations in the early 70s. These experiments involved trying new combinations or creating a *soup* which is a random assortment of living cells with a particular density within the grid[5]. Creating a soup allows researchers to watch the evolution to try and pick out new objects.

The elements as described in this section will play an important role in building logic gates and circuits. Objects such as *gliders* and *guns* have been studied since

the 1970s while other elements, such as the splitter, are built up from these more basic building blocks. This section will go over the most basic elements required for a circuit before the section on building the logic gates.

4.1.1 Gliders

The most basic requirement to build a logic gate will be a wire, or a stream of information. This is where the notion of spaceships within Conway's Game of Life will play a pivotal role. A spaceship is an object that moves across the universe as defined below

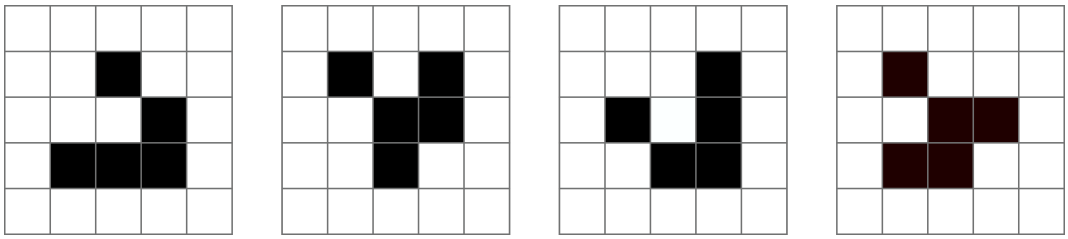
Definition 29. A *spaceship* is a finite pattern that returns to its original state after a certain number of generations in a differenty location.

The number of generations it takes to appear in its original orientation is the object's period. While many different spaceships exist with varying speeds, periods, and sizes the spaceships that this dissertation focuses on is the smallest spaceship known as the *glider*.

Definition 30. A *glider* is a particular spaceship consisting of 5 cells that travels diagonally across the grid with a period of 4 generations depicted in 4.1

The glider was discovered in 1970 by Richard K. Guy and remains important in the study in the Game of Life[15]. Gliders are always made up of 5 living cells and have four orientations as seen in 4.1. Their small and simple design allows for easy creation and streams of gliders can be produced by glider guns. Collisions of gliders also play an important role in the creation of more complicated structures. Gliders can be used to create different, sometimes very complicated, objects in so called glider synthesis through collisions.

Figure 4.1: Glider Orientations



4.1.2 Guns

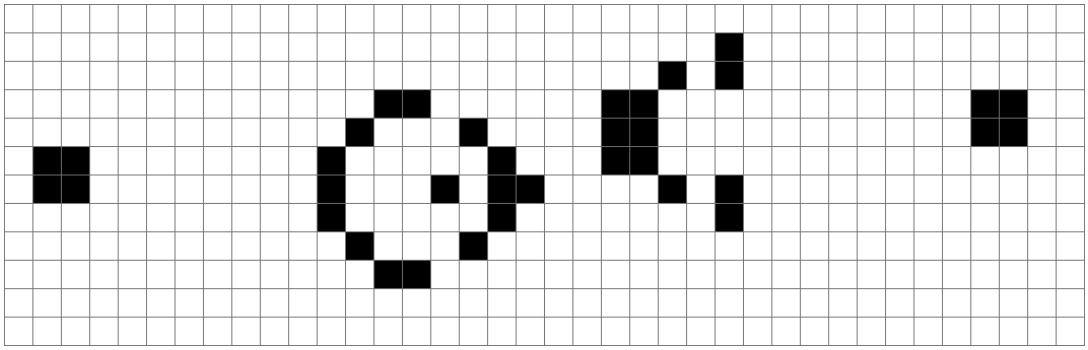
Another component needed is a method to create gliders on demand. This is where the idea of a gun is important.

Definition 31. A gun is a stationary finite pattern that emits spaceships at a regular interval.

The spaceships are emitted from a particular point on the gun called the barrel at a regular interval, creating a stream of spaceships in a direction. There are two periods to a glider gun, the emitting period and the period of the gun itself. While these periods may differ, though the period of the gun must be a multiple of the emitting period, many guns have been found with their two periods being the same.

The Gosper Glider Gun, found in 1970 by Bill Gosper was the first such gun discovered[15]. It has a gun period and emitting period of 30, creating a glider every 30 generations in a diagonal direction. It is composed of two oscillating elements, called queen bee shuttles, between two blocks as seen in figure 4.2. This element was pivotal in the first proofs of completeness as it allowed for the generation of a stream of information.

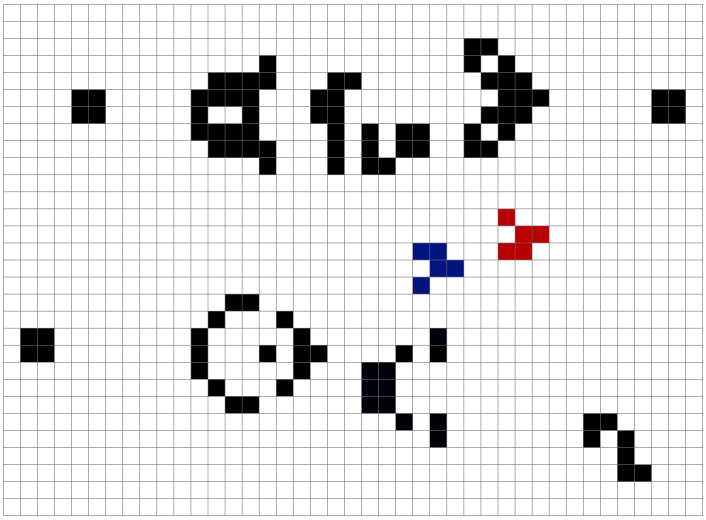
Figure 4.2: Gosper Glider Gun



Differing configurations of glider guns allow for a variety in the period of the stream. The Gosper glider gun emits a glider once every 30 generations. This places the gliders too close to one another for many of the constructions that will be used later in the dissertation. We can create a period 60 gun, eliminating this problem. This gun is composed by a Gosper glider gun and a modified Gosper glider gun beneath it to thin the stream as desired shown in figure 4.3[15]. The top portion of the finite pattern is comprised of a traditional Gosper glider gun which produces a stream with period 30 depicted as the red glider in 4.3. The lower portion creates gliders depicted in blue that destroy every other red glider using a collision. This produces a stream of gliders depicted in red with a period of 60. These glider guns

form the basis for nearly all of the advanced structures needed for building logic gates and circuits.

Figure 4.3: Period-60 Glider Gun



4.1.3 Lanes and Glider Streams Collisions

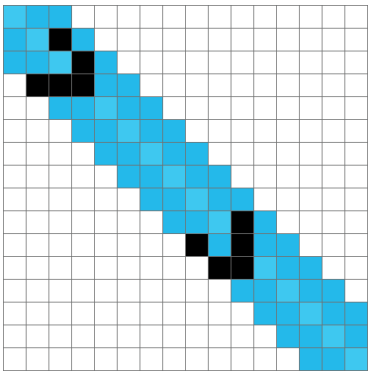
The method for creating streams of gliders has been laid out but more details on these streams will be presented here. First a lane is defined as follows:

Definition 32. A *lane* is the cells within the bounding box of a spaceship along the path it takes.

Figure 4.4 shows a glider stream of period 30 with the lane of the gliders highlighted in blue. Lanes exist with or without gliders. If a glider gun is not producing gliders the lane will continue to extend where gliders can be present once it begins producing again. Wherever the lane is clear a spaceship can move undisturbed. Lanes will represent wires carrying information later in the constructions. Thus when a lane is empty we say it is representing 0 and when the lane has a stream of gliders it is representing 1.

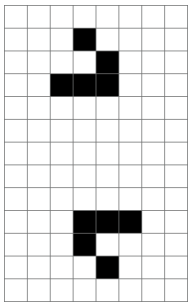
When lanes containing streams of gliders touch one another there are a few types of glider collisions possible. Here is an outline of the most common collisions that will be seen in these constructions. There are 72 unique possible glider collisions between two gliders[15]. While many of these collisions produce undesirable results synchronizing and lining up streams correctly allow us to avoid these undesirable outcomes in favor or two particular favoured outcomes creating nothing or creating a block.

Figure 4.4: Glider Lane



The first desirable collision is the right angle collision which produces nothing as a result. There are 6 such collisions but not all of these collisions take the same amount of generations to complete. The smallest such collision requires five generations from the point the bounding boxes overlap to complete. The gliders are lined up as in figure 4.5. The collision itself plays out shown in 4.6 producing no residue after only 5 generations.

Figure 4.5: Glider Nothing Producing Collision Alignment



This collision is useful for completely destroying, or blocking, streams of the same period. If two streams of both period 30 or both period 60 collide in this fashion both streams will be completely destroyed. This type of collision can also thin a stream. If a stream of period 30 collides in this fashion with a stream of period 60 then the period 60 stream destroys every other glider in the stream of period 30. This leaves us with a single stream of period 60.

The next type of useful collision is the collision that leaves behind a block. Obtaining a collision of this type is also a matter of lining up the stream correctly shown in 4.7. The smallest right angle collision of this type also takes 5 generations to complete as shown in 4.8.

This collision is more important when working with streams of differing periods. Since a block is left in the middle of the stream the next glider that comes will get

Figure 4.6: Glider Nothing Producing Collision

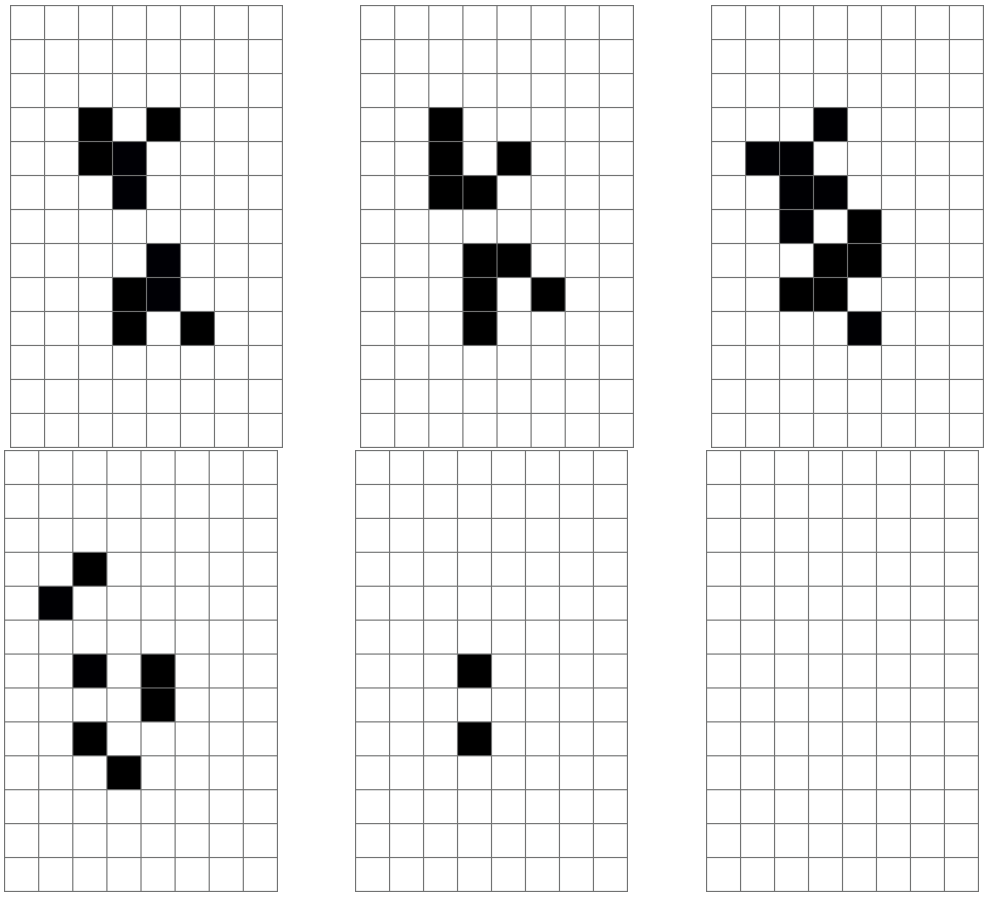
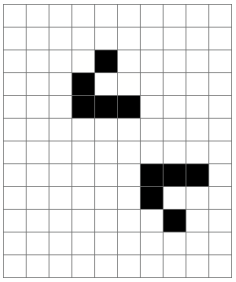


Figure 4.7: Glider Block Producing Collision Alignment



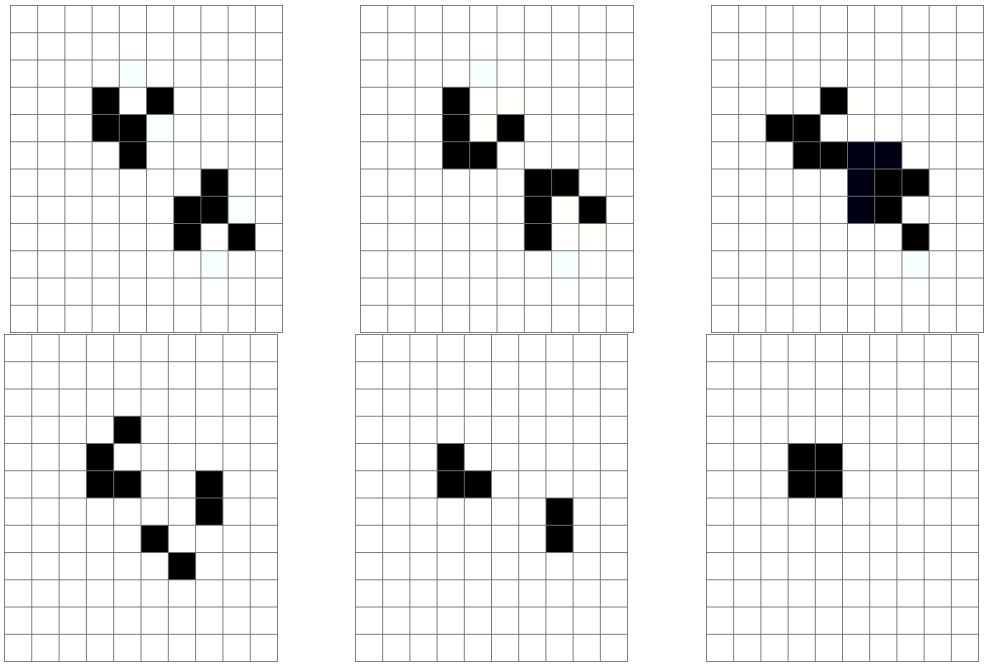
destroyed on the block. Thus a period 60 stream can completely destroy a period 30 stream with double the gliders using this type of collision.

4.1.4 Eaters

An eater is an object, typically a still life, which "eats" an incoming object.

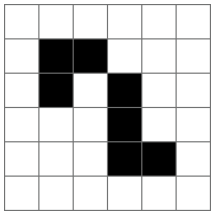
Definition 33. An *eater* is a finite pattern which after a collision returns to its original state after a finite number of generations.

Figure 4.8: Glider Block Producing Collision



This means it will destroy an object such as a glider that collides with it but the eater itself will remain intact. These objects have a period of recovery which is the amount of generations after a collision that the object takes to return to its original state. These objects are useful for stopping a stream of gliders as they can survive many collisions one after another. One of the earliest known eater seen in 4.9 takes four generations to recover from a glider collision and therefore can be used to stop streams emitted from the Gosper Glider Gun which emits a glider every 30 generations[15].

Figure 4.9: Eater 1



4.1.5 Changing Direction

Typically when dealing with logic circuits the notion of a wire's direction is taken for granted. When building a circuit in the physical world wires can be bent into a direction and on paper drawing a bend is not an impossible task. Streams of gliders

can move in one of four directions along the diagonals. The issue here is the streams are only able to move in a straight line without bends or turns. Four directions is enough for building a circuit but to be able to connect the various streams in meaningful ways the ability to change the direction of a stream is required.

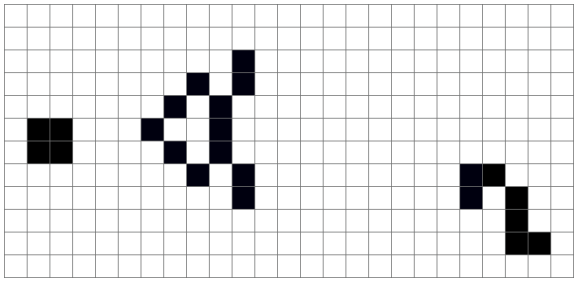
There are many ways to change the direction of gliders. The most basic way is using an finite pattern called a reflector.

Definition 34. A *reflector* is a finite pattern which takes an input spaceship and outputs the same spaceship at a different angle of trajectory without suffering permanent damage.

Reflectors that specifically reflect gliders have become well documented. There are some key considerations when constructing a reflector besides the spaceship it accepts. Two important considerations are the recovery time and the period. The recovery time needs to be smaller than the period of the gun that is outputting a stream so the reflector is ready to accept the next glider. The period of the reflector must also divide evenly into the period of the glider gun producing the stream. Reflectors with oscillating components need to be in the correct point in its period to accept a glider properly. By having the period of the reflector divide evenly we ensure the gliders always collide with the reflector in the correct point in its period.

One such reflector that is both simple and meets the criteria above is called the *buckaroo*. The buckaroo was first found by David Buckingham in the 1970s[15]. It is a reflector with a period of 30 and a recovery time of 30. Thus it is able to reflect a glider every 30 generations. The buckaroo specifically can reflect a glider by 90° in either direction or 180° back on itself. Though reflecting gliders 180° will be of little use in these constructions being able to reflect 90° will prove to be incredibly useful in circuit construction.

Figure 4.10: Buckaroo Reflector



The buckaroo functions by producing what is known as a *banana spark* with the glider as it collides with the buckaroo. A *spark* is a finite pattern that dies after finite generations. The banana spark is commonly used to reorient gliders. The

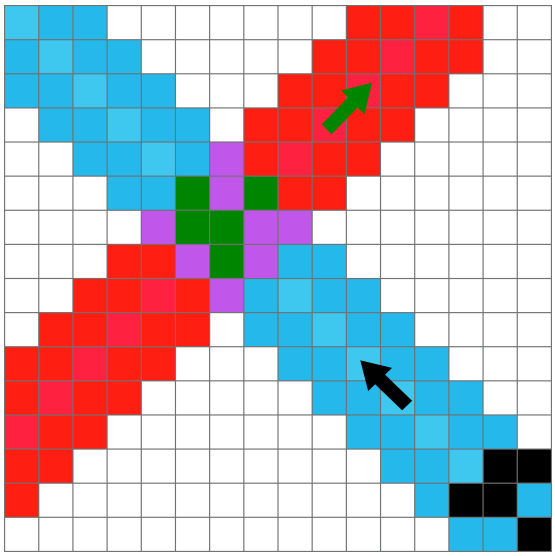
buckaroo produces these sparks through collisions that dissipate unless it interacts with a glider. Thus the buckaroo is able to function even without a stream of gliders colliding with it.

4.1.6 Crossing Wires

When building circuits in Conway's Game of Life there is only two dimensions that the circuits can be built in. The wires in our circuits, the streams of gliders, need to be able to cross one another without corrupting the stream. In conventional cases the wires can jump over one another but without that third dimension to operate in this is impossible. To solve this problem the period of the stream can can be used to bypass the need for a jump.

Having streams of period 60 allows us to have enough space between gliders so crossing lanes can be offset enough to allow the gliders to miss one another. Shown in figure 4.11 are two period 60 glider streams, with the black gliders travelling in the north west direction along the blue lane and the green glider travelling in north east direction along the red lane, the gliders can be timed to miss one another while crossing lanes. With the gliders travelling at a speed of $\frac{c}{4}$ it moves one cell in the x direction and one cell in the y direction every 4 generations. The centers of each glider are displaced at least eight cells in both the x and y direction in this configuration. With the bounding box extending two cells out from the center in each direction this leaves four cells between the bounding boxes which is enough at the speed of $\frac{c}{4}$ to prevent a collision.

Figure 4.11: Crossing Glider Lanes



4.1.7 Splitter

Another key feature of circuits is the ability to split a wire so that it may connect to multiple gates or outputs later in the circuit. Since our wires consist of streams of gliders this poses more of a challenge. Since glider streams with period 60 are required for most of the constructions, peeling away gliders and placing them into a new lane is not an option and creating new gliders will become necessary. This is where an object, a splitter, comes into play.

Definition 35. A *splitter* is a finite pattern that when it collides with a spaceship produces two or more spaceships.

Many splitters are constructed using patterns such as the herschel, as seen in figure 4.12, which produce multiple gliders after some number of generations. These processes have been shown to have a period as low as 30 generations but often have a high recovery time. Thus a different method of duplicating gliders is necessary for completing this circuitry.

Figure 4.12: Herschel Finite Pattern

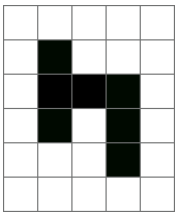
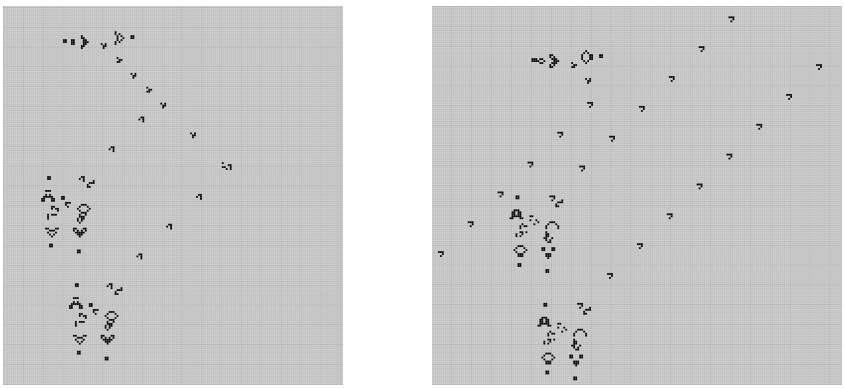


Figure 4.13: Glider Splitter

(a) Splitter Without Input Gliders (b) Splitter With Input Gliders



A simpler solution would be to construct a splitter out of glider guns. The streams containing information have a period of 60. A period 30 Gosper glider gun

can block two period 60 streams as it has double the gliders than the individual streams as seen in figure 4.13a. An incoming period 60 stream can utilize a glider collision that leaves behind debris, such as a block, as discussed in the collision section above thus blocking the period 30 stream shown in figure 4.13b. This interaction is important for building the circuits required for memory storage and is relatively simple. This construction only relies on 3 glider guns and outputs streams that are in the same point in their period as seen by the glider orientations in figure 4.13b making it easier to combine components in the future.

4.2 NOR Gate Construction

The construction of the NOR gate within Conway's Game of Life is relatively simple. Various logic gates have been implemented in Conway's Game of Life, their design choices often involve the medium of information transfer, the configuration of inputs, their complexity among other things. Rennard constructed, AND, OR, and NOT gates using lanes of gliders that interacted at 90 deg angles. Rennard aslo utilized a finite pattern referred to as a *stopper* to stop the flow of gliders and stop the flow of information. The design below uses only collisions to stop the flow of information limiting the number of components needed in the NOR gate simplifying the construction.

The NOR gate construction can be achieved with a single glidergun, and to keep the stream moving in the same direction, a buckaroo. This gate is constructed to take two inputs and have a single output. The main idea behind the construction is the collision of gliders. The only time the gate has an output is when there are no incoming streams. Using a single glider gun this output stream is created perpendicular to the inputs. The buckaroo aligns the output to match the direction of the inputs. Between the glider gun and the buckaroo are the input streams, only one of which is needed to block the output stream.

Theorem 36. *Conway's Game of Life can simulate logical NOR.*

Proof. Let glider lanes represent lanes of information with gliders in the lane representing 1 and no gliders in the lane representing 0. Consider three cases:
Case 1: The input is $(0, 0)$ the output stream is not blocked from exiting the gate thus outputting a value of 1 as seen in figure 4.14
Case 2: The input is $(0, 1)$ then the stream of gliders representing 1 blocks the output stream thus outputting a value 0 as seen in figure 4.15. Without loss of generality the same is true for an input of $(1, 0)$ as seen in figure 4.16.

Case 3: The input is $(1, 1)$ the output stream then similarly to case 2 will be blocked by the upper input thus outputting a value of 0 as seen in figure 4.17.
These cases show the truth values to be identical to logical NOR thus this gate simulates NOR. □

Figure 4.14: NOR Gate With input $(0, 0)$

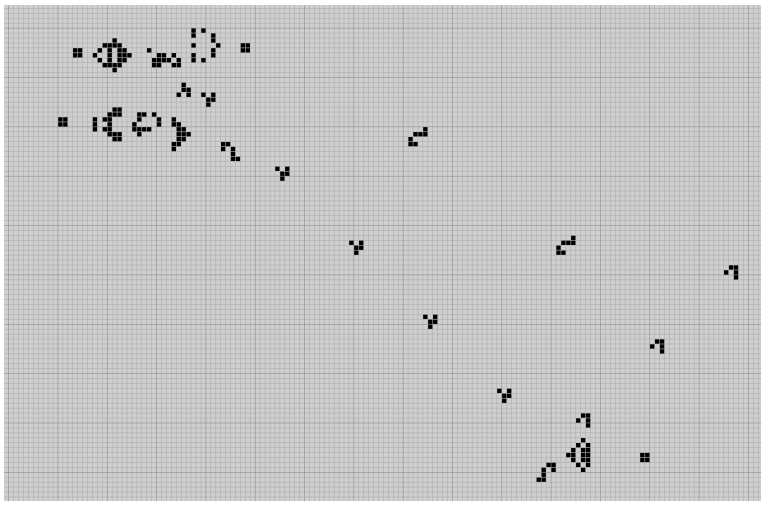
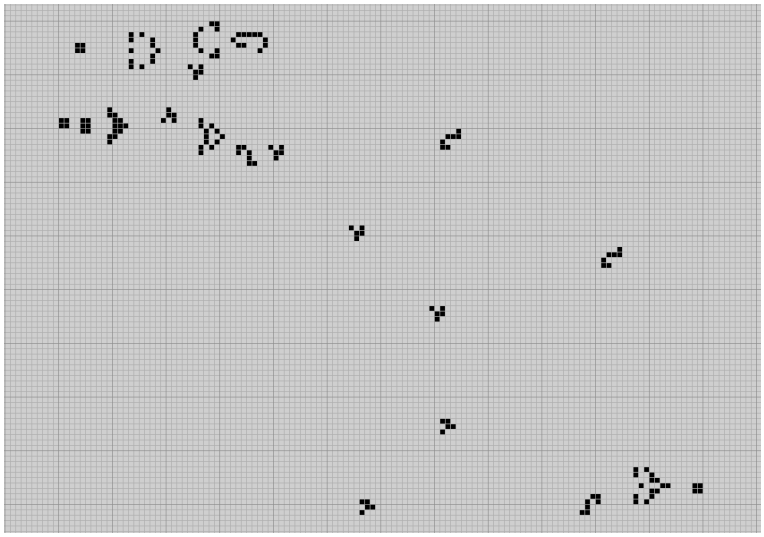


Figure 4.15: NOR Gate With input $(0, 1)$



Since the collisions of only a single input stream is enough to stop the output stream this gate is incredibly simple. Two main components, the buckaroo and 60-period glider gun, are required in building the gate is a matter of lining up the lanes of input gliders for the correct collision and lining up the buckaroo with the output stream.

Figure 4.16: NOR Gate With input $(1, 0)$

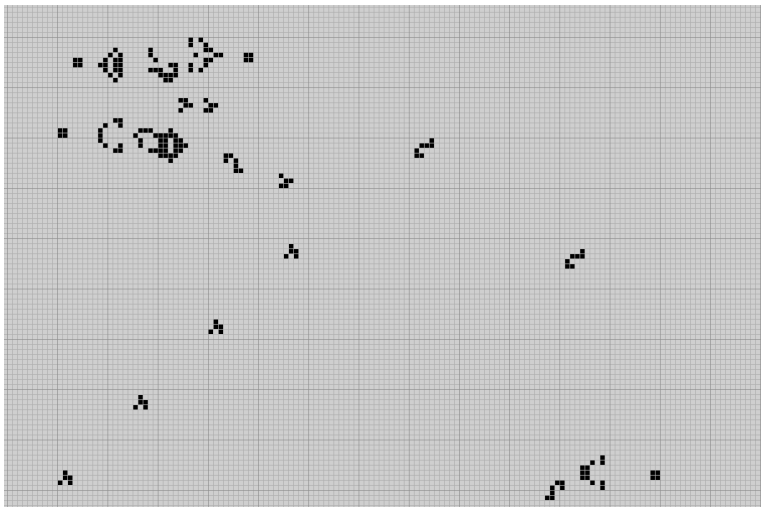
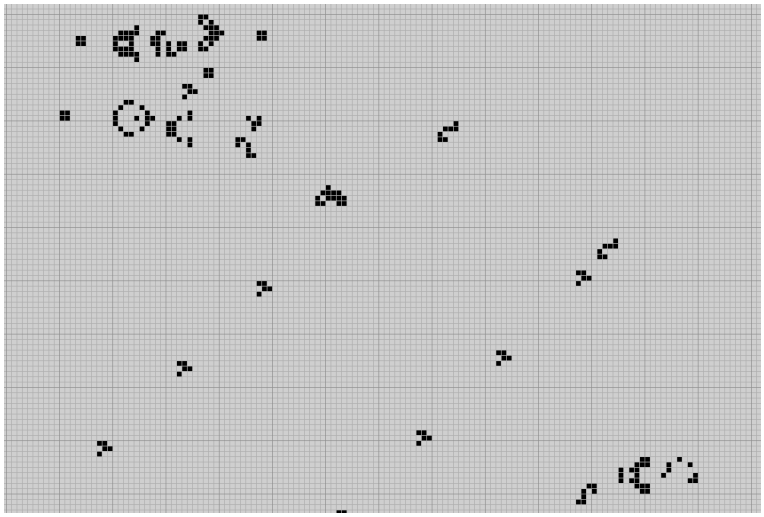


Figure 4.17: NOR Gate With input $(1, 1)$



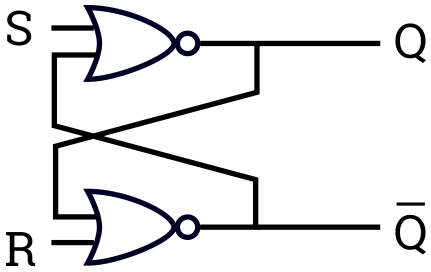
4.3 Memory

With the NOR gate the possibility of storing values takes shape. This ability to store information is the next step in achieving completeness in the system. The next sections will begin to utilize some abstraction when discussing the ideas of memory and completeness. These larger components and systems are built up from the same small components discussed previously and as the system gets more complicated as does the discussion of how it works. This dissertation's focus is on achieving Turing completeness rather than simulating a Turing machine and thus abstraction will help in the final sections.

To prove that Conway's Game of Life is indeed Turing complete we will require the ability to store and look up values arbitrarily and this will require one final

object, the *SR NOR Latch*. This latch is a type of Set-Reset latch which can store one of two states denoted Q and $\bar{Q}$. While the $\bar{Q}$ state is on if S is pulsed then the latch is set to Q, likewise if Q state is on and R is pulsed the latch is reset to $\bar{Q}$. This is due to the latche's feedback on itself, as seen in 4.18. The case where $S = R = 1$ is the forbidden state as it would force $Q = \neg Q$. This problem can be bypassed through clever circuit design which is beyond the scope of this dissertation. This will allow for storage of either 1 or 0 which is enough to store values in binary.

Figure 4.18: SR NOR Latch



Constructing this latch in Conway's Game of Life is a matter of connecting two NOR gates as constructed previously in the correct orientation. The latch will take two input glider streams, one for set and one for reset, and have two output streams for Q or $\bar{Q}$ only one of which will have an output stream at one time. The construction will require the usage of two NOR gates along side two splitters and two buckaroo.

These latches can be set up into registers. These registers hold a single integer and are uniquely addressed. The storage of integers can be achieved by storing binary numbers with each latch representing either a 1 or 0. Addressing the registers is done by referencinig the location within the universe by building lanes to the register. Circuits can send signals to different parts of the universe using the lanes and these signals can be controlled by gates guiding the signal down the correct lanes to the final register.

4.4 Completeness

For a system to be Turing complete it needs to be able to simulate a universal Turing machine. Previously discussed proofs of completeness of cellular automata included the usage of the Post tag system and the ability to simulate a universal Turing machine within the cellular automata. This section will outline how using the components described previously Conway's Game of Life can simulate a Turing-equivalent system, the random access machine.

The first step in showing that the random access machine can be constructed will be to show that combinational logic circuits capable of boolean algebra are possible.

Theorem 37. *Conway's Game of Life can simulate a combinational logic circuit capable of Boolean Algebra.*

Proof. A combinational logic circuit is composed of 3 components: G a labled acyclic graph with a set of E edge-labels and L labels for the verticies. As constructed above glider lanes can represent edges with labels $\{0, 1\}$. The ability to cross glider streams and change directions allows for all graphs to be represented. This can be restricted to only acyclic graphs. The NOR gates act as verticies in the labels L and constitute a functionally complete set of logic gates. Thus Conway's Game of Life can simulate a combinational logic circuit capable of Boolean Algebra. □

Next step will be showing that it is possible to build a sequential logic circuit.

Theorem 38. *Conway's Game of Life can simulate sequential logic circuits.*

Proof. A sequential logic circuit similarly to the combinational logic circuit is composed of 3 components: G a labled graph with a set of E edge-labels and L labels for the verticies. The difference is circuits are allowed to have cycles. Glider lanes represent edges with labels $\{0, 1\}$ where 0 is no gliders present and 1 is gliders are present. The ability to cross glider streams and change directions allows for all graphs to be represented. The NOR gates act as verticies in the labels L and constitute a functionally complete set of logic gates. Thus Conway's Game of Life can simulate a combinational logic circuit. □

Now that some basic circuitry is set up within Conway's Game of Life the idea of memory within the system becomes important. Sequential logic circuits have the ability to take previous output as an input, and thus allow for memory in a system. Thus we can prove the following theorem.

Theorem 39. *Conway's Game of Life can simulate a Finite State Machine.*

Proof. By the previous theorem and Theorem 15 since Conway's Game of Life can simulate a sequential logic circuit Conway's Game of Life can simulate a finite state machine. □

Registers are addressed memory locations, since Conway's Game of Life has locations within the grid these can be addressed by having lanes that take the information to particular locations on the grid as described in section 4.3. This is the final component needed to show Conway's Game of Life is Turing complete. The next theorem gives us the final construction needed.

Theorem 40. *Conway's Game of Life can simulate a random access machine.*

Proof. Conway's Game of Life can simulate a finite state machine by the previous theorem. Since Conway's Game of Life exists on an infinite universe an infinite number of registers can be created and accessed through the use of sequential logic circuits. Thus Conway's Game of Life can simulate a random access machine. □

The simulation of a random access machine is the final step we need. Since a random access machine is Turing equivalent the following corollary follows.

Corollary 41. *Conway's Game of Life is Turing complete.*

4.5 Speed and Efficiency

Though completeness can be achieved in Conway's Game of Life and the use of streams of gliders and constructions of logic gates give are an analog to modern computers, the reality is this construction would create a very inefficient computer. For example, Rennard's two-bit binary adder took 2.5 minutes to run through its calculation[13]. This dissertation does not focus on complexity theory and therefore has focused on the computations that are possible but this section will comment on the speed and efficiency.

In all these constructions described, the information carrying component are the gliders. Gliders are only a single type of spaceship that is possible to construct within Conway's Game of Life. The glider has a speed of $\frac{c}{4}$ or $\frac{1}{4}$ the speed of light within Conway's Game of Life. Though no spaceship can travel faster diagonally, there exist spaceships that can travel at $\frac{c}{2}$ in the cardinal directions. Constructions that are even faster exist with transmission methods that can move at the speed of light. Jason Summers created a telegraph or a fintite pattern capable of transmitting information at the speed of light. This construction is incredibly complex and requires substantial setup and time to reset. Gliders having much more flexibilty allow for simpler constructions at the cost of slower information transfer speeds.

Random access machines as a model of computation relies on the ability to use information stored in an infinite amount of registers. In theory these registers have a lookup that should be uniform, but within Conway's Game of Life registers further away can take a substantial time to lookup. Reckhow and Cook showed that a RAM can simulate a Turing machine T in $O(T \times l(T))$ where the function $l(n)$ represents the storage time[4]. While the RAM model is typically faster than a Turing machine if the storage is inefficient these gains are not felt in the final computation. Thus while the system is Turing complete the speed and efficiency is lacking.

Chapter 5

Conclusion

Mathematicians in the early 20th century asked which problems could mathematics possibly solve. This complex question gave rise to the entire field of computability theory and led to a revolution in problem solving. These deep rooted mathematical ideas laid the groundwork for modern computing and an ability to put mathematics to use on an even larger scale to solve problems. There is a deep well of literature in the area of computability but the study of particular systems in regards to their universality is often sparse. Conway's Game of Life is an incredibly well studied Turing complete system but the literature still needs expanding.

By utilizing formal models of computation a system can be evaluated for how computationally powerful it is and whether fulfing Turing complete computation is possible. These were the steps taken to build the powerful computers we see today and these foundational ideas are still being utilized in evalutating new systems in computation and complexity of algorithms. These ideas allow us to see the viability of new computing models such as Quantum Computing.

Conway's Game of Life represents a sufficiently complex system that can be utilized to perform advanced computations. These examples help us study the limits of computing and design of computing systems. Alan Perlis coined the term Turing Tarpit to describe a system "in which everything is possible but nothing of interest is easy"[9]. Conway's Game of Life may be an example of a Turing Tarpit, a Turing complete system which is difficult to learn and use. Even though the viability of the system to solve problems is not good there is active interest in Conway's Game of Life as a model of computing from people continuing to study the complex stuctures that arise to recent attempts to build modern computing architecture using only Conway's Game of Life [2].

Conway's Game of Life and cellular automata in general are interesting systems to study. Their complex behaviour lends them to be useful in studying a large va-

riety of phenomenon including computing. Cellular automata have applications in modelling biological and physical systems as well as aiding in education about these systems. Completeness of certain automata is still widely studied with applications in collision based computing[18]. Beyond the classical notion of cellular automata new inventions such as *Quantum Dot Automata* contribute to an ever growing literature studying the ties of universality and cellular automata. Using a visual system such as Conway's Game of Life to study complex topics, can help students and the public alike understand concepts and apply them. This dissertation has taught me a lot about Turing completeness, computation, and automata. I hope this dissertation will serve as a clear explanation of why a system like Conway's Game of Life is Turing complete.

Bibliography

[1] Alvy Ray Smith III. Simple Computation-Universal Cellular Spaces. *Journal of the Association for Computing Machinery*, 18(3):339–353, July 1971.

[2] Nicholas Carlini. Digital Logic Gates on Conway's Game of Life - Part 1, April 2020.

[3] Matthew Cook. Universality in Elementary Cellular Automata. *Complex Systems*, page 40, 2004.

[4] Stephen A Cook and Robert A Reckhow. Time Bounded Random Access Machines. page 22, August 1972.

[5] R.Wm. Gosper. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena*, 10(1-2):75–80, January 1984.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, Boston, 3rd ed edition, 2007. OCLC: ocm69013079.

[7] Jarkko Kari. Universal pattern generation by cellular automata. *Theoretical Computer Science*, 429:180–184, April 2012.

[8] Marvin Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Englewood Cliffs, NJ, 1967. OCLC: 899037781.

[9] Alan Perlis. Epigrams on Programming. *SIGPLAN Notices*, 17(9):7–13, September 1982.

[10] Emil L. Post. Formal Reductions of the General Combinatorial Decision Problem. *American Journal of Mathematics*, 65(2):197, April 1943.

[11] Rendell. A Turing Machine In Conway's Game Life.

[12] Paul Rendell. A Universal Turing Machine in Conway's Game of Life. In *2011 International Conference on High Performance Computing & Simulation*, pages 764–772, Istanbul, Turkey, July 2011. IEEE.

[13] Jean-Philippe Rennard. Implementation of Logical Functions in the Game of Life. In *Collision-Based Computing*, pages 491–512. Springer London, London, 2002.

[14] John E. Savage. *Models of computation: exploring the power of computing.* Addison Wesley, Reading, Mass, 1998.

[15] Stephen A. Silver, Dave Greene, and David Bell. Life Lexicon, July 2018.

[16] Alan Turing. On Computable Numbers with and Application to the Entscheidungsproblem. November 1936.

[17] Stephen Wolfram. *A new kind of science.* Wolfram Media, Champaign, IL, 2002.

[18] Liang Zhang and Andrew Adamatzky. Collision-based implementation of a two-bit adder in excitable cellular automaton. *Chaos, Solitons & Fractals*, 41(3):1191–1200, August 2009.