## Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Author</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erratum</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>The Idea of Functionalism</td>
<td>Ricardo Santos</td>
<td>4</td>
</tr>
<tr>
<td>Using Turing Machines to Simulate Computer Programming</td>
<td>Nabil Ahlhauser</td>
<td>8</td>
</tr>
<tr>
<td>Prim’s Universal Turing Machine</td>
<td>Dustin McAfee</td>
<td>10</td>
</tr>
<tr>
<td>On The Possibility of a Turing Machine Implementation in the Fallout 4 Building System</td>
<td>Nicholas Romano</td>
<td>13</td>
</tr>
<tr>
<td>Parsing using Pushdown Automaton</td>
<td>Jacob Sutton</td>
<td>16</td>
</tr>
<tr>
<td>Turing Machines: ASCII code, interpretation, encryption, and printing</td>
<td>Jake Mason</td>
<td>19</td>
</tr>
<tr>
<td>Basic Propositional Logic Statements through Turing Machines</td>
<td>Matthew Jenkins</td>
<td>23</td>
</tr>
<tr>
<td>Prime Number Determination with a Multiple Tape Turing Machine using Martin Davis Notation</td>
<td>Michael Kaufman</td>
<td>27</td>
</tr>
<tr>
<td>E-Z Sequence</td>
<td>Kayla Brown</td>
<td>30</td>
</tr>
<tr>
<td>Computing Game Theory Outcomes with Turing Machines</td>
<td>Tanner Curren</td>
<td>33</td>
</tr>
<tr>
<td>On Comparators and Turing Machines</td>
<td>Will Marttala</td>
<td>36</td>
</tr>
</tbody>
</table>
Erratum

While nearly perfect, the previous issue of MCTOC contained one small error. The name of one of our authors, Dustin McAfee, was misspelled as "Dustin McAffee". This error is corrected within the table of contents for the present issue. The editor would like to extend his sincerest apology to all members of the McAfee family.
The Idea of Functionalism

Ricardo Santos
Division of Computer Science & Mathematics
Maryville College
Ricardo.santos@my.maryvillecollege.edu

Abstract—Functionalism is the doctrine that believes that the mind could be realized in a wide variety of physical states. It views mental states such as pain, as software and the brain as the hardware implementing the software. This doctrine has its roots from Aristotle times, but it was furtherly developed by Alan Turing, Hilary Putnam among others. Functionalism is currently one of the most popular theories of mind. Is connected to the idea that Artificial Intelligence will one day be programmed so that it can challenge human intelligence. Functionalists only care about the function being performed. They only need to describe the inputs and outputs. Consequently, they believe that the mind can be successfully simulated by a Turing Machine or computer program and realized in a physical state such as a computer. There is a great debate in whether simulating the mind is possible or not. The different concepts and arguments against and in favor of functionalism are discussed.

Keywords—computer functionalism; functionalism; Alan Turing; Turing Test, The Chinese Argument; machine functionalism; psychofunctionalism;

I. INTRODUCTION

Functionalism, is a doctrine that what makes something a mental state of a particular type does not depend on its internal constitution, but rather on the function it performs [2]. This doctrine roots from Aristotle’s conception of the soul. It became fully articulated in the last century with the works of Alan Turing, Hilary Putnam among others. Functionalism was born as an opposition or alternative to the identity theory which states that the mind is a physical state that can be realized in a computer or hardware. According to the identity theory, for every mental state there exists a unique mental-chemical state of the brain such that a living organism can be in that state if and only if it is in that physical state [8]. The identity theory was rejected by Putnam who argued that for living organisms to be in the same mental state, they will first have to be in the same physical state which we know is impossible [8]. The identity theory views the brain as the computer and the mental states as the physical states of the computer while functionalism views the mental states as the software. Both theories believe in the idea that the mind can be realized in another physical state such as a computer, but with exception that functionalism views mind as software rather than hardware. Functionalists are only concerned about the function being performed by the mind rather on how the mind performs its functions. Can mental states be multiply realized in a wide range of different physical states? Can we create a Turing Machine that can simulate the mind? We will go over the views of functionalists and try to see if their theory is strong enough to support the idea that the mind could be realized in a computer and simulated by a Turing Machine or computer program. According to functionalists, since mental states are functional rather than physical, mental states can be multiply realized in a wide variety of physical states.

II. APPROACH

A. Features of Functionalism

In this subsection, we will define and explore the basic concepts of functionalism. This includes mental states and multiply realized. Examples are provided to make clearer the beliefs of functionalists.

a) Mental States

According to functionalism, mental states serve as an intermediary between causes, which consists of certain sensory inputs, and effects which consists of behavioral outputs [1]. Examples of mental states include pain, desire, beliefs, fears etc. Functionalists are only concerned with the role being performed by the mental state. They are able to describe the functions of mental states without knowing what those mental states are composed of. We can compare them to black boxes, were we only need to describe the inputs and outputs. For example, let’s suppose that we have three DVD players from three different brands. The DVD players might be composed of different materials or their architecture might be different, but they will all be capable of playing a CD of mp3 format. We do not exactly know how are they composed or how do they play a CD. We only care if they are capable of playing our CD. The same happens with functionalists, they only care for the fulfillment of the function. Functionalists are refer as physicalists, those who believe that everything that exists is composed of matter [2].

b) Multiply Realized

Although functionalists are physicalists, they claim that it is at least imaginable that mental states such as pain can be realized in something other than molecules. This claim leads to the belief that mental states can be realized in a wide variety of physical states or that they could be multiply realized [2]. For example, a man that is hit by a car will be in great pain. If a deer is hit by a car on a highway, is the deer going to feel pain? Remember that according to functionalists, we do not care what is the deer’s brain made of or how pain is produced, we need only to care about the inputs and outputs. Since the deer was also hit by a car in motion and the deer feels pain then the pain felt by the man and the deer fulfills the same causal role. Recall that the identity theory claims that for every mental state there exists a unique mental-chemical state of the brain such
that a living organism can be in that state if and only if that
living organism is in that physical state [8]. Let’s go back to
the example of the deer and the man hit by the car and think
about it in terms of the identity theory. Then feeling pain can
only be realized by human beings because for a deer to feel
pain that deer will have to be in the same physical state. In
other words, the deer will have to be in a human body to feel
pain, but we know that is not the case. From this point of view,
it makes more sense to think that mental states can be realized
in a wide range of physical states than in a single physical
state.

B. Variations of Functionalism

In this section, we will introduce the ideas of Hilary
Putnam, one of the founders of functionalism. We will then go
over two well-known variations of functionalism: machine
functionalism and psychofunctionalism.

a) Hilary Putnam and Functionalism

Hilary Putnam is an American philosopher, mathematician,
and computer scientist who is well-known for being one of the
founders of functionalism and his arguments against the
identity theory [7]. As mentioned earlier, the identity theory
views the brain as the computer and mental states as physical
states of the computer. Putnam argued that it is impossible for
organisms to be in the same state they will first have to be in
the same physical state which we know is impossible. For
example, pain in humans is triggered by C-fibers, but animals
do not have C-fibers. So animals will not be able to feel pain,
but we know that animals can feel pain [1].

Putnam became one of the founders of functionalism after
he published a paper titled Minds and Machines in 1960. He
theorized that mental states are functional properties. In other
words, the mind is like a computer program. Later on, in the
late 1980s, Putnam changed his mind and argued against
functionalism. This was primarily to the difficulty that
computational theories have of explaining some intuitions
with respect to the externalism of mental content. This means
that in order to be in an intentional mental state, it is necessary
to be related to the environment in the right way.

b) Machine Functionalists

Functionalist see the mind as the software and the brain as
the hardware. Recall that mental states are causal functional
states that perform a distinctive role. According to machine
functionalists, the mind can be physically realized in a Turing
Machine. In other words, they claim that we could build a
Turing Machine complex enough that can simulate the mind.
We could give it a set of instructions so that it can perform the
same functions the mind can perform. Two machines can be in
the same state if and only if they both realize the same Turing
Machine. So two systems cannot share a mental state,
they must have all mental states in common. For instance, a
human could not share pain with a deer because they do not
have all mental states in common. But we know that is not the
case, we can share some mental states even though we do not
have them all in common.

As a consequence of the restrictions of machine
functionalist, this theory was further developed into
Psychofunctionalism. Psychofunctionalism lies between the
common sense and the scientific approach [10]. Block, its
founder, claims that machine functionalism tends to be
dependent of what is observable. Without this restriction,
psychofunctionalism creates a more realistic account of
mental states. It also avoids giving consciousness to things
that cannot have a mind. The main problem is that it is hard
for psychofunctionalism to keep a balance between the
generalized nature of multiply realized and human exclusive
scientific observation [9]. Also as consequence of the
restrictions of machine functionalism, many computer
scientists and philosophers have moved into this version of
functionalism.

c) Psychofunctionalism

Machine Functionalism suggests that for two systems to be
in the same mental state, they both have to realize the same
Turing Machine. So two systems cannot share a mental state,
they must have all mental states in common. For instance, a
human could not share pain with a deer because they do not
have all mental states in common. But we know that is not the
case, we can share some mental states even though we do not
have them all in common.

As a consequence of the restrictions of machine
functionalist, this theory was further developed into
Psychofunctionalism. Psychofunctionalism lies between the
common sense and the scientific approach [10]. Block, its
founder, claims that machine functionalism tends to be
dependent of what is observable. Without this restriction,
psychofunctionalism creates a more realistic account of
mental states. It also avoids giving consciousness to things
that cannot have a mind. The main problem is that it is hard
for psychofunctionalism to keep a balance between the
generalized nature of multiply realized and human exclusive
scientific observation [9]. Also as consequence of the
restrictions of machine functionalism, many computer
scientists and philosophers have moved into this version of
functionalism.

C. Arguments

We will introduce some of the most famous arguments
against and in favor of functionalism. These arguments
include the Turing Test, the Chinese Room Argument, and
Qualia. The advantage of this, is that we will be able to see the
beliefs of both sides of the argument.

a) The Turing Test

In 1950, Alan Turing wrote a paper titled Computing
Machinery and Intelligence when he was working at a
computer laboratory in Manchester University [5]. This paper
included answers to specific objections of artificial
intelligence, suggestions on how we will arrive at artificial
intelligence, and the Turing Test. He proposed that it is
possible to construct a Turing Machine (or computer) complex
enough that can simulate the mind and that can also trick
another human being that it is a real human being not a
machine.

Turing put forward his idea of an “imitation game”, in
which a human being and a computer would be interrogated
by another human being. The interrogator does not know
which one is the computer and which one is the human. The
whole communication is held entirely by text messages
(keyboard terminal). Turing argued that if the interrogator
could not distinguish between the machine and the human,
then it is unreasonable to think that a computer is not
intelligent. The Test provides an objective criterion of what is
being discussed with the necessity of the computer to assert
something false to apply the ‘imitation’ of a human being [6].
Of course, the machine will have to be complex enough or
programmed with a great amount of information about
language syntax, grammar, culture etc. In other words, the
machine will have to be prepared with the enough information
to be able to answer literal and non-literal questions. For example, these computer will have to be able to answer to questions about feelings and things involving the use of the five senses (i.e. How do you feel? What do you smell?). The test was later on refer as the Turing Test for intelligence.

b) The Chinese Room Argument

One of the most famous arguments against functionalism is The Chinese Room Argument. This argument was proposed by American philosopher John Searle. Imagine a person, P, that does not know how to speak Chinese. This person is confined in a room with a set of rules to exchange strings of Chinese characters for another string of Chinese characters [6].

Once the person becomes familiar with the set of rules, then the person will start ‘outputting’ Chinese expressions when given as input a Chinese expression. If this person is viewed from the outside, it will seem that this person actually speaks Chinese. Recall that functionalists only pay attention to the input and output, they do not actually care how the output is performed. The same happens with a computer program or Turing Machine, it will act in some way depending on the input since it is govern by a set of rules. Critics of functionalism argue that the mind is more than a Turing Machine that pushes symbols depending on the given input on the tape. So they believe that the Turing Test is an invalid test for mentality. Searle suggests that no amount of syntactic symbol-pushing will generate meaning [6]. In other words, no matter how complex a computer program or a Turing Machine is, it is not possible for them to simulate the mind since the mind is more than pushing symbols.

c) Qualia

Critics of functionalism claim that functionalists fail to take into account the qualitative aspect of the mind. Qualia is simply a name given by philosophers. Qualia are experiential properties of sensations, feelings, perceptions, and thoughts. Critics of functionalist argue that brain states cause consciousness, but these things do not tell us anything about subject experiences [4]. For example, what happens if we see an orange leaf? Does our brain finds and enters into our orange state? The answer is no, critics of functionalism believe that a computer is uncapable of having true subjective experiences (without being programmed).

Consider the event of having an identical zombie twin, adapted from Chalmers’s The Conscious Mind. This creature is physical identical to you. Let’s suppose that it is a fall afternoon and you and your twin zombie go for a walk in the woods and both of you stop to see the orange, red, and brown leaves of the trees in the woods. You are amazed of the beauty of nature. What about your twin zombie? Is he/she going to feel the same way? The zombie is going to be processing the same information, reacting the same way to inputs, and modifying its internal configuration appropriately [4]. The problem is that your zombie twin is not experiencing real conscious experience, is just following a set of rules.

III. Conclusion

Functionalism is a doctrine that believes that the mind can be realized in a Turing Machine or computer program. We have seen how functionalists believe the brain to be the hardware that implements the mind which is seen as the software. As a consequence of the works of Alan Turing and Hilary Putnam, many computer scientists have continue to develop numerous versions of this doctrine. Many even believe that by the year 2030, artificial intelligence at human level will be reached [5]. This means that there would exist computers that would not only be able to think, but also will be exceed or surpass the capacity of human beings.

There are several versions of functionalism, in this paper, we briefly explored two of them. Machine functionalists believe that the mind can be realized in a Turing Machine or computer program. A very well-known argument use to support this thesis is the Turing Test. The Turing Test states that we can build a computer complex enough that will be able to simulate the mind and trick another human that is not a machine, but a human being. Turing puts it in terms of an ‘imitation game’. The idea that we can have an interrogator who will ask questions to the human and the machine via keyboard terminals to determine which one is the human. According to Turing, the interrogator will be unable to distinguish between the computer and the human, thus, we have a computer complex enough that can successfully simulate a mind. We also talked about psychofunctionalism which is similar to machine functionalism, but without the restriction of observation by pure common sense.

Of course, there are many critics that argue against the functionalist doctrine. One of the most famous arguments is the Chinese Room Argument. We saw that from the outside, it seems that the person inside the room seems to speak Chinese. The person inside the room is being provided with input (Chinese expressions) and is outputting another Chinese expression which is what we expect. Functionalists will only describe the input and output, but not how is done. If we look closely, the person inside the room, that person is only following a set of instructions. Searle concludes that the mind is more complex than pushing symbols and thus, we cannot create a Turing Machine complex enough to simulate the human mind.

Other critics of functionalism argue that functionalism ignores the qualitative aspect of the mind. Even if we have a Turing Machine that could simulate the mind, it will unsuccessfully do so. This is because the machine is not really experiencing consciousness, but it is just systematically responding to input.

Some critics argue that Turing Machines and computers only follow a set of instructions and they do not really experience consciousness. In other words, Turing Machines have be programmed to perform the same functions the mind does. If we think about human nature, for example, we are also ‘programmed’ to think, feel, and believe by a greater being. From this point of view, Turing Machines could successfully simulate the human mind.
It is hard to argue against functionalism because we know that Turing Machines, for example, can compute things that we can also compute and even more than that. Some critics argue that Turing Machines can only be in one state at a time and humans can be in multiple states (i.e. stressed and tired at the same time) so it is impossible for Turing Machines to simulate the mind. We know this is not the case, since we could have a multi-tape Turing Machine that could be in multiple states and simulate the mind successfully. It is still a debate whether that level of artificial intelligence will ever be reached or not. To continue this work, we could develop our own version of functionalism with a multi-tape machine that can simulate the mind.

REFERENCES

   URL: http://www.csun.edu/~tab2595/12_Causal_Theoretical_Functionalism.pdf

   URL: http://www.mind.ilstu.edu/curriculum/functionalism_intro/functionalism_intro.php

   URL: http://plato.stanford.edu/entries/functionalism/

   URL: https://www3.nd.edu/~jspeaks/courses/2006-7/20208/absent-qualia.pdf

   URL: http://www.turing.org.uk/scrapbook/test.html

   URL: http://www.philosophyonline.co.uk/oldsite/pom/pom_functionalism__qualia.htm


   URL: http://www.iep.utm.edu/identity/

   URL: https://platoscinema.wordpress.com/tag/psychofunctionalism/

[10] Unknown, Block’s Troubles with Functionalism,
    URL: http://www.jimpryor.net/teaching/courses/mind/notes/block.html
Abstract— While the subject of computers emulating Turing machines is a well-known one, the subject of Turing machines simulating computer programming languages is a less studied subject. Consequently, this paper examines how Turing machines can replicate data types, variables, the if keyword, the while keyword, and functions. Lastly, the paper briefly examines how the Turing machine could halt in a similar way to how programs end.

Keywords—Turing machines; Computers; Programming

I. INTRODUCTION

Alan Turing wrote a paper about Turing machines in 1936. In it, he discussed how the machines could be used to theorize computations. His paper is considered by some to be the birth of the study of computer science.

While computers have evolved away from Turing machines, we can still use them to emulate Turing machines. For example, the website https://turingmachinesimulator.com/ provides a place where someone can input their own configurations and run the machine [1]. Additionally, Todd Veldhuizen examined how some of the C++ templates are Turing complete [2]. However, the paper points out that there will probably never be a rigorous proof. As a result of the fact that computers can emulate Turing machines, modern computers are considered Turing Complete.

While a lot of work has been done to examine how computers can simulate Turing machines, less work has been done on the subject of Turing machines simulating computers. This is partially due to the fact that emulating computers is a bit more complicated. For one, there are multiple ways to emulate them. For example, this paper will focus on the higher level aspects of the computer, namely programming languages. This paper will discuss how to mirror some of the commonly used programming language instructions on a Turing machine.

II. APPROACH

We will proceed by defining some of the commonly used programming structures and then giving a high level description of how to create a Turing machine that can read those instructions.

A. Data Types

In programming, there are multiple different types of data including Booleans, integers, and strings. These data types can be similarly represented on a Turing machine by inserting a special character symbol representing the data type before the characters that would normally be used to represent the data.

B. Variables

Variables are used in programming to store some data. When used, they recall the type of the data and the data itself. The data can then be used for a variety of things. For example it can be manipulated and restored, it can be compared to other data, or it can just be read.

By default, the use of variables is not an explicit feature of a Turing machine. That said, there are multiple methods for implementing variables on a Turing machine. We will examine the use of an additional Turing machine tape. The second tape can store the variables on it. The tape can simply contain a separator character symbol to indicate a new variable. After the separator, the variable name, type, and its value can all be stored with a blank to separate each of them.

Now that we have described how variables are stored, we will explore how variables can be called. When a variable is used on the primary tape, the Turing machine can search through the second tape until it finds the corresponding variable name on the second tape. From there, the machine can read or modify the variable. Should the variable not exist, a new one can be created at the end of the tape. Lastly, should a variable’s memory need to be expanded, the machine can generate more room by moving all the characters right. While incredibly inefficient, this method allows for an infinite number of variables.

C. The If Keyword

The if keyword compares two expressions. After evaluating the expressions, the machine decides whether or not to execute the statements that follow. If the comparison is true, the machine runs the statements. If the comparison is false, those statements are ignored.

To represent this control structure on a Turing machine, we will begin by having a symbol represent the keyword if. From there, the machine can read an expression, the
comparison symbol, and then the second expression. To help with the recognition of these parts, the machine can have a blank in between each section. Lastly, the machine can use two symbols to signify where the statements begin and end.

In order to compare the two expressions, a comparison system has to be created. If the two expressions are Booleans, this problem can simply be answered by comparing the truth values. If the expressions are integers, this can be done by comparing their values from left to right. This character by character comparison will allow the machine to evaluate any potential inequality by discovering if the values are larger, smaller, or the same. If strings are compared, a character by character comparison can also be applied to check for equality. In programming, inequalities can be used to compare a wide variety of data types. However, this paper is limited in scope and will only examine the above mentioned ones.

When comparing the two characters, the machine can use Gödel numbers. Most languages are composed of finite characters and thus the machine can enumerate each character using a Gödel number. Even if the language is infinite, it will be countably infinite and thus can still be enumerated and then compared.

In order to assist with the process of nesting instructions, all of the symbols used in this control structure should contain subscripts. Subscripts allow the machine to keep track of where the control structure begins and ends. Using subscripts ensures that the machine ignores any other statements and only finds and evaluates the correct one.

D. The While Keyword

The while keyword checks a condition before executing some statements. The process is then repeated until the condition is false. This process can be replicated on a Turing machine. Since the while keyword checks a condition, we can use the same logic of the if keyword to evaluate the truth of the condition. That said, we will need to come up with unique characters to symbolize the beginning and ending of the while condition.

The while keyword also differs from the if keyword in the way that it finishes executing. Once the Turing machine has finished executing the statements, the machine must recheck the condition to evaluate its truth. This can be done by searching for the beginning while symbol character and then reevaluating the condition.

E. Functions

In programming, functions are used to easily reuse code and also to help with readability. Functions break the code into smaller pieces. Turing machines can simulate all of this. In programming, functions are frequently added to the end of the program. We can copy this technique by placing the function at the end of the tape. When the Turing Machine reads a function’s name, it can search the rest of the tape for the function keyword followed by the functions name. From there the function can execute.

In order to return to the correct spot once the function has finished, we can create a variable that stores the location of where the function was called. By enumerating each character on the tape, we can save the location as an integer value relative to the beginning of the tape.

Now that we have examined some of the programming instructions, we must examine when the machine will halt. To insure that this Turing machine eventually halts, we can insert a halting symbol. This symbol functions similarly to the return 0 statement in C++ as it will allow us to force the machine to halt where we specify. This specification has the additional advantage of letting us examine if the machine halted correctly. Frequently, this symbol will be read as the last thing on the tape before the functions. While this symbol exists, there is no guarantee that the machine will halt. By placing the symbol in an if statement that evaluates to false or by creating an infinite loop, it is possible to never reach the symbol.

III. Conclusion

We have shown some of the ways a Turing machine can emulate a computer’s instructions. We have examined how the computer uses data types, variables, the if statement, the while statement, and functions and then discussed how a Turing machine could emulate them. Lastly, we examined when the machine could halt.

While this paper has shown a machine that can emulate a computer, there are many way that it can be improved. For example, we have only covered a few programming concepts. Many other programming instructions, such as the for loop, are not included. Additionally, while we have represented these instructions on the tape, they are not represented in the most efficient way.

While this Turing machine will probably never really be used, it does demonstrate some important principles. The fact that computers can be emulated on a Turing machine shows how Turing machines are still relevant. Excluding the space restriction imposed on physical computers, Turing machines and computers are computationally equivalent. Ultimately, both of them will be here for a while.


Prim’s Universal Turing Machine

Dustin McAfee
Maryville College, Computer Science and Mathematics Double Major

Abstract—Universal Turing Machines can be used for graphing algorithms. In the scope of this paper specifically, Prim’s Algorithm for finding a Minimal Spanning Tree can be utilized by a Universal Turing Machine. The input will need to follow a certain syntax that will be discussed, as well as a modification to Alan Turing’s Universal Turing Machine using similar notation to On Computable Numbers, with an Application to the Entscheidungsproblem by Alan Turing. Graphing syntax for Turing Machines can be generalized to other algorithms as well.

I. INTRODUCTION

A Universal Turing Machine, given other Turing Machine algorithms, is able to complete many (as many as given) computable formulae, as long as it is inputted in the language of a Turing Machine. Some graphing algorithms, such as Prim’s Algorithm for finding Minimum Weight Spanning Trees for Graphs, can be represented by Turing Machines. Alan Turing wrote his paper on the Universal Turing Machine, which uses skeleton table notation which will be utilized in the formation of our Prim’s Algorithm Turing Machine. A slight extension to Turing’s Universal Turing Machine will be defined in a way that takes on this iterative graphing algorithm, as well as a generalization of how this extension could be used for multiple graphing algorithms. Cohen’s Introduction to Computer Theory also gives useful Turing Machine examples, such as deletion and insertion, that will be utilized. Our algorithm will only work on undirected graphs, and modification for directed graphs is out of the scope of this paper.

II. UNIVERSAL TURING MACHINE

Alan Turing describes how to set up the single tape Turing Machine for the Universal Turing Machine using the Standard Description (S.D.) of the Turing Machine in which it will execute, as well as the execution of the Turing Machine from its S.D. [1]. Therefore, this paper will not go into detail on how the Universal Turing Machine processes our algorithm; instead, we will start with a working copy of Alan Turing’s Universal Turing Machine, extend it, and build a Turing Machine for Prim’s algorithm that can be transcribed onto the tape of our Universal Turing Machine. This Universal Turing Machine will have two tapes: The standard as described in Turing’s paper, and the input graph whose syntax will be discussed in a later section. This is not necessary, as the second tape can be concatenated with a delimiter onto our first tape, but this simplifies our model for the reader. This requires two heads of the Universal Turing Machine, in which execution of our algorithm will require the first head to shuffle back and forth through our modified Universal Turing Machine, causing the second head to carry out instruction on our graph input tape.

A. Input Syntax

A graph may have one to some very large finite amount of vertices, which will be on the input tape for Prim’s algorithm. Specifically, $\Sigma = \{\cdot, \alpha, a, b, c, d, \ldots, a, b, c, \ldots, (, 1, \$, where all lowercase letters (and duplications) are unique vertices. The input starts with the list of vertices with periods in-between (F and E squares) followed by two $\alpha$’s (one on an F square and another on an E square), the list of edges and weights denoted by triplets and ordered by weight, such as $ab \ 111$ as an edge in-between $a$ and $b$ with weight 2, and a delimiter denoted $. An example graph such as below will have an input series (where periods represent E squares) denoted as: $a.b.c.d.e.f.\alpha\alpha (a.b.1.1.1). (a.c.1.1.1). (b.d.1.1.1.1). (c.e.1.1.1.1.1). (d.f.1.1.1.1.1.1). (e.f.1.1.1.1.1.1). (b.c.1.1.1.1.1.1.1). (d.e.1.1.1.1.1.1.1.1).$ followed by an infinity of blank symbols we will denote $B$.

B. Modification

This is where a bit of abstraction comes in. Alan Turing’s Universal Turing Machine had the S.D. at the beginning of the input tape (just like our first tape in our Universal Turing Machine), and it had instructions on how to carry out the instructions of the S.D. it was simulating [1]. Before running anything, our modified version of Alan Turing’s machine will change every $X$ in the S.D. of the machine it is simulating to a beginning symbol in the second tape, in which $X$ represents an arbitrarily picked vertex, and the beginning symbol is a vertex in the input graph, and is deleted from the beginning of the input after it is used. However, in Turing’s notation all symbols must be represented by $D$ followed be a repetition of $C$’s, but it is clear that there is a 1-1 correspondence between the finite amount of vertices and the natural numbers (Turing’s representation of the tape’s symbols). So, our modification will have to insert or delete a certain number of $C$ symbols in the S.D. of the first tape, based on what representation is chosen for $X$ and the symbols of the vertices on the second tape. These Turing Machines have been built as examples in chapter 24 of Cohen’s book, so they will not be explicitly redefined here [2]. After the $X$ symbols have been replaced and the original symbol in the input tape deleted (or blanked out), the S.D. is simulated, and then the next vertex is then substituted
PRIM’S ALGORITHM TURING MACHINE, 30 NOVEMBER, 2015

III. PRIM’S TURING MACHINE

Here is where Turing’s skeleton tables will be used; this paper will rely on a few already produced tables from Turing’s paper [1]. First, here is Prim’s algorithm in pseudo-code for a graph \( G \) with \( n \) vertices, and \( T \) MST (originally no edges and all vertices):

Initialization;

\[
\text{for } i = 1 \text{ to } n \text{ do } \\
\quad e := \text{an edge of minimum weight incident to a vertex in } T \text{ and not forming a circuit in } T \text{ if added to } T; \\
\quad T := T \text{ added } e; \\
\quad i = i + 1; \\
\text{return } T; \\
\]

Algorithm 1: Prim’s Algorithm

We will define our first skeleton table as \( f(C, B, a) \) similar to the one in Turing’s paper [1].

\[
\begin{align*}
\text{m-config.} & \quad \text{Symbol Behavior} & \quad \text{Final m-config.} \\
f(C, B, a) & \begin{cases} 
B & L \\
not B & R \\
\alpha & L \\
not \alpha & L \\
a & N \\
\end{cases} & \begin{cases} 
f_1(C, B, a) \\
f_2(C, B, a) \\
f_3(C, B, a) \\
\end{cases} \\
f_1(C, B, a) & \begin{cases} 
\alpha & L \\
not \alpha & L \\
a & N \\
\end{cases} \\
f_2(C, B, a) & \begin{cases} 
nota & R \\
B & R \\
a & N \\
\end{cases} \\
f_3(C, B, a) & \begin{cases} 
nota & R \\
B & R \\
\end{cases}
\end{align*}
\]

This will find the first symbol \( a \) after \( \alpha \) in the input tape for the graph. Let Prim’s Turing Machine have arbitrary vertex denoted \( X \). The Prim’s Turing Machine is as follows (with \( B \) as terminal state for failure):

\[
f(A, B, X)
\]

This marks everything in the parenthesis to the left (on \( E \) squares): 1’s with \( \beta \) and vertices with \( \delta \). The reasoning for this follows Turing’s m-configuration table for \( ce_3(B, \alpha, \beta, \gamma) \), which copies down at the end of the tape the first symbols marked \( \alpha \), then those marked \( \beta \), and finally those marked \( \gamma \); it also then erases the symbols \( \alpha, \beta, \gamma \) [1]. Only two of these markers will be used in our algorithm. If there does not exist \( X \) after \( \$ \) in the input tape, then \( A_3 \) will call \( ce_3 \) to copy the 1’s, followed by our two vertices. Otherwise, we will go into the m-configuration \( A_4 \). Our algorithm will also erase everything in the first edge containing \( X \) after copying, which will be called right after \( ce_3 \) or \( A_4 \) and will not be written as its own skeleton table since it closely resembles pre-built skeleton tables from above and in Turing’s paper. Instead, for the sake of simplicity, we will include it after \( ce_3 \) or after \( A_4 \) and it will be denoted \( e(X) \). For \( A_4 \), an edge has already been copied passed \( \$ \) in the tape, an edge has already been erased, and there is another edge that is marked with \( \delta \) and \( \beta \). Also note here that the head is pointing at \( \$ \) in the input tape.

\[
\begin{align*}
\text{m-config.} & \quad \text{Symbol Behavior} & \quad \text{Final m-config.} \\
A_3 & \begin{cases} 
B & N \\
\end{cases} & \begin{cases} 
X & N \quad ce_3 \\
\end{cases} \\
\text{else } R & A_4 & \begin{cases} 
A_3 \\
\end{cases}
\end{align*}
\]

\( A_4 \) is essentially the same as \( ce_3 \) except that it goes from the right to the left when copying elements, stopping at the delimiter, \( \$ \). For simplicity, the entire table will not be given here, but is left to an exercise to the reader. This is the greedy portion of the algorithm: the minimum weight will be copied from every edge containing \( X \) (which is substituted by our Universal Turing Machine one by one for each vertex). Since the edges are ordered from least to greatest by weight, there will be only enough room for the smallest amount of successive 1’s per vertex. First the 1’s are copied, and if the delimiter is not reached, then the vertices are copied at the end of the tape. \( e(X) \) is then called which deletes the first edge in the tape containing \( X \). This ensures that there are no circuits in
our MST. Repeating until all vertices are enumerated ensures the Minimal Spanning Tree for Graph \( G \).

**IV. Proof of Correctness**

Assume that our spanning tree is not minimal. Then there exists an edge from \( u \) to \( v \), denoted \( e \), in our spanning tree that is not in the minimal spanning tree. Say this edge is during our \( k^{th} \) iteration of our Universal Turing Machine. Then there also exists a path from \( u \) to \( v \), denoted \( P \), such that \( P \) is in the MST. Let \( e' \) be an edge in \( P \) such that only one endpoint is in the tree generated by the \((k-1)^{th}\) iteration of our Universal Turing Machine. If the weight corresponding to \( e' \) is less than the wait corresponding to \( e \), then our algorithm would’ve chosen \( e' \) on its \( k^{th} \) iteration. This means that the weight of \( e' \) is greater or equal to the weight of \( e \). Substituting \( e \) for \( e' \) gives less or equal weight than our spanning tree, and repeating this process for all iterations yields the Minimal Spanning Tree for a graph at any instance of Prim’s Algorithm.

Our ordered input for our graph is necessary for the greedy portion of the algorithm: This ensures that we will only be picking the least weight edges. No circuits may form either, since all edges are deleted after being copied. Therefore, our Prim’s Turing Machine calculates the Minimal Spanning Tree for any given undirected graph.

**V. Applications**

Our version of the Universal Turing Machine can, given the number of repetitions, loop through an iterative algorithm by replacing symbols in the S.D. before each run. This can be used for any calculable finite iterative algorithm. Also, we have defined a syntax for graphing algorithms in a Universal Turing Machine. Prim’s Algorithm was a sort of example, but one could just as effectively create a Universal Turing Machine that could do other graphing algorithms such as Dijkstra’s algorithm. Any computable graphing algorithm can now be effectively written as a Universal Turing Machine, and can be visibly demonstrated in this way.

**VI. Conclusion**

Graphing algorithms are computable by Universal Turing Machines; specifically, with a simple iterative modification to Alan Turing’s Universal Machine, Prim’s Algorithm can be computed by a Universal Turing Machine. This creates an entire syntax for graphing algorithms in Turing Machines. The efficiency could definitely be improved, since for every vertex the description of the Turing Machine increases drastically. In the future, Turing Machines simulating graphing algorithms may one day calculate the efficiency of other Turing Machines (which, in general, can be represented by graphs).

**VII. References**

On The Possibility of a Turing Machine Implementation in the Fallout 4 Building System

Nicholas Romano

Abstract—In the trend of modern video games having sandbox modes where the player can create virtual structures limited only by their imagination, their computer memory, and the set of the building tools within the game, the thought experiment of the simulation of Universal Turing Machines (UTM) within these games has become more common in the minds of computationally-inclined players. Through this thinking and a degree of experimentation, the author conjectures that one of the most recent such games to be released, Fallout 4, has a set of in-game tools that may have an allowing complexity to implement a TM but not necessarily a UTM. Such conjectures are not definitive and need to be explored further.

I. Introduction

As a well-known example of a video game which can implement a UTM, Mojang’s 2011 release, Minecraft, allows for a large multitude of possibilities due to its granularity and large set of possible ”blocks”. Some of these blocks, called redstone, emulate electronic circuitry and can be combined with in-game switches and other device blocks to create logic gates, much like those in a physical computer processor. Unsurprisingly, these gates can then be further combined to create a UTM, proving Minecraft’s Turing completeness assuming infinite space and time [1]. The implications of the existence of a Minecraft UTM is that, again assuming infinite space and time, Minecraft redstone circuits can solve any computable problem.

Possibly inspired by Minecraft, Bethesda’s 2015 release, Fallout 4, includes crafting and building systems which are inconsequential to the game at large. That is, these systems of gathering resources from the wasteland and using them to create structures and devices are not designed to be the primary objective of the player, and therefore they are not as developed as the systems present in Minecraft. As a result, the crafting and building systems of Fallout 4 are of a lower complexity than Minecraft due to the fewer tools available. Importantly, we should ask ourselves whether, despite its reduced complexity, the building system in Fallout 4 can also implement TMs and whether it is also Turing complete.

II. Wasteland Toolbox

To begin, we must explore the tools available to the player. Unlike Minecraft, Fallout 4 strives for realistic immersion in the game world and therefore contains elements of realism that permeate to the structures the player can build. The following sections follow the given names of the relevant devices. Later, we will ignore building costs, power production requirements, and spatial requirements as these considerations are not inherently useful for the conceptual systems we will create.

A. Generators

Fallout 4 devices require power to function, and this power is, obviously, generated by the generators. There are a multitude of these generators available to build, each providing a varying production of power. Each generator can be hooked to the other generators, adding their productions together.

B. Wires

Functioning in a manner similar to real-life wires and circuits, Fallout 4 wires instantly carry power from one device to another. Any device can have any amount of wires connected to it, and if one wire has power on it, then all of the wires connected have power unless the connection is a switch in the off position.

C. Manual Switches

As their name suggests, manual switches require player input to manipulate them into the on or off positions. When wires are connected, their power is not transferred to each other unless this switch is in the on position. Specific examples of these switches are ordinary activatable hand switches, pressure plates the player can stand on, or lasers the player can step through.

D. Terminals

Some devices have multiple settings that can be changed. Terminals allow the player to change these settings by connecting the device and the terminal by wires and then accessing the settings on the terminal screen.

E. Delayed On/Off Switches

Delayed switches act like the manual switches do but without player input. After a time of being powered set by a terminal, the switch will either turn on or off depending on the type of delayed switch the player chooses.

F. Interval Switches

Much like delayed switches, interval switches are set to turn on or off depending on a terminal-defined time. The difference with interval switches is that they will return to their initial state if they retain their power, performing a loop of switching actions.
G. Power Counters

Perhaps the most complex of the devices are the power counters, which are switches which contain up to ten internal states, with state zero as the initial state. Whenever the counter is in state zero, the switch is considered on. Every time power is transferred to the power counter, the state is increased by one until it reaches its max state, set by a terminal. Once it reaches its max state, the counter will return back its initial state, zero.

H. Lightboxes

Used as the output of wires, lightboxes are simply boxes which emit light whenever a wire which connects them is powered. Incidentally, a terminal can set the display of the lightbox to one of seven colors.

III. REQUIREMENTS AND SCOPE

Before we continue, it is important to consider the implications of the following work we will do. In order to build a classical TM, the implementing system must have an infinite tape of data containing a finite language, a movable read/write head to view and manipulate the data, and a finite internal set of states which contain the instructions to be carried out. In order to build a UTM, a specific TM which can take in the standard description of another TM and perform the actions within the inner TM must be constructed in the implementing system. To be clear and avoid fallaciousness, our work will not show whether TMs or UTMs can be constructed in Fallout 4, but we can show some evidence for the positive claim and build on this towards a possible proof in the future. Nevertheless, the evidence we will presently show is that a tape containing a finite language and a head which reads input from a movable point on this tape can be constructed.

IV. THE WASTELAND TAPE MACHINE

Our implementation of the relevant parts of a turing machine, called the Wasteland Tape Machine (WTM), will combine a few of the parts mentioned in section II to create a finite tape and a head which can read from a position on the tape. First, there will be two switches on separate wires from the generator, named the Safety and Counter switches. Connected to the Safety switch will be the cells of the tape in the form of more ordinary switches. Furthermore, there are as many power counters as cell switches with each switch wired into one power counter. The Counter switch is in turn wired to each of the power counters. The Safety switch is necessary to ensure power does not get transferred through the cell switches and increment the power counters, which are delicately incremented in succession so that only one is in the initial state at one discrete time step. See the diagram below for an implementation of this machine with three cell switches, marked with Xs. A lightbox can be connected to the cylindrical power counters to act as an output for whatever cell the "head" power counter is seeing.

A. Execution

The execution of the WTM involves first setting the cell switches to their on or off positions, representing the symbols 1 or 0, while the Safety and Counter switches are set to off. For example, if we wanted a 3-symbol tape to represent 101, we would turn the four switches on, off, and on, respectively.

Next, to ensure we are starting at the right place, the counter switch is toggled as many times as needed to get the power counter in the initial state zero on the relevant starting head, leaving the switch on when it arrives. Once it does, we switch on the Safety switch and then switch off the Counter switch. At this point, the output of the head power counter matches with the state of the cell switch it rests on. Thus, information can be read from the tape.

To move the tape to the right, we must toggle the Safety switch off then toggle the Counter switch on. This moves the power counters up one time step, and the counter in the initial state is now one to the right. To complete the cycle of movement, we must now toggle the Safety switch on and then the Counter switch off. These actions return us to our beginning switch configuration, save for the power counter states (see table below). The head power counter is now reading information from the cell one right to the beginning cell.

<table>
<thead>
<tr>
<th>Switch</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
<th>t₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety</td>
<td>On</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
</tr>
<tr>
<td>Counter</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
<td>On</td>
</tr>
</tbody>
</table>

B. Possible improvements and limitations

Of course, the WTM is not perfect. A human player is needed to keep the machine running, and even then it is pretty simple. While the present description of the machine only has movement to the right of the tape, movement to the left can be simulated due to the circular nature of the power counters. That is, instead of moving left once, the machine can move n−1 times right, where n is the number of cells in the machine.

Furthermore, the existing actualizations of the Safety and Counter switches are within the class of manual switches, and it is conceivable that they can be replaced with the interval switches if cleverly timed. Doing so would cut out the need of a human player mindlessly toggling the switches in the right order to make the movements he or she wants.

As mentioned before, a lightbox can be attached to all of the power counters so it can output whatever signal the cell switch corresponding to the head power counter is emitting.
This acts as a read from the tape of cell switches, but no case has been made for a write to the tape. This is due to the need for either a human player or a complex set of in-game circuits which are currently not supported. The latter has not been proven not to exist, but neither has it been proven to exist, so the writing ability of the WTM is, currently, nonexistent. This fact obviously distinguishes it from a TM, but we are reminded that the WTM is meant to be a part of a whole to help show that a TM within Fallout 4 is not necessarily impossible.

V. Conclusion

Our work with the WTM, while not definite, shows that an effort can be made to make something resembling a part of a TM. Until it can be shown that a TM can be made, and with it a UTM, the question of Fallout 4’s building system being Turing complete is still a debatable subject. Future work to be done includes implementing structures past the simple reading of a tape, such as automatically writing based on internal states and the extension to a practically infinite tape instead of being limited to 10 due to the maximum states of the power counters. Perhaps if a form of logical circuit can be made without direct human intervention, much greater improvements can be made.

References

Abstract—The viability of an expression is important in computation, Pushdown Automata can be used to verify the acceptability of expressions by making sure it holds to a set of grammatical rules. We will be creating a pushdown automaton which can verify basic arithmetic expressions, which will reject uncomputable expressions and accept computable ones. This application of pushdown automaton is used in many parsers, especially in programming languages and in devices like calculators. This paper will also address the different kinds of parsers that can be made from pushdown automata and the benefits of each.

I. INTRODUCTION

Pushdown Automata have less computation use than, for example, a Turing Machine, however PDA are quite useful for things like language parsing. Like a Turing Machine, PDAs read input from a tape, however, it consumes the input as it reads it and uses a stack with the operations Push, to push things onto the stack, and Pop, to pull things out of the stack. When the tape ends or the PDA moves into a terminal state then the PDA will either reject or accept the language. Typically if the state is non-empty at the end of the tape then the language is rejected. Parsing is done by having a set of grammatic rules, Context-Free Grammar, which can be used to verify the validity of the expression. If the expression is adheres to the rules than it’s acceptable language. This type of parsing is often used in programming languages. We will demonstrate the use in parsing arithmetic expressions and determining their validity using PDA to either reject or accept the string.

II. PDA, PARSING AND CONTEXT-FREE GRAMMAR

A. Description of PDAs

A pushdown automaton is 7-tuple \( \{ Q, \Sigma, \Gamma, \delta, q_0, Z_1, F \} \)

- \( Q \) is the total set of possible states
- \( \Sigma \) is the acceptable input alphabet
- \( \Gamma \) is the pushdown stack alphabet
- \( \delta \) is the transitional function
- \( q_0 \) is the initial state
- \( Z_1 \) is the bottom of pushdown stack symbol
- \( F \) is the total possible of final states

Our transition function \( \delta \) is set up as:

(current state \( q_n \), input character, pop character or \( \varepsilon \), next state \( q_m \), push character)

Example:

We will construct a PDA for a very basic example of a language where there is some number of A’s followed by an equal number of B’s.

\( A^nB^n \in w \quad \text{where} \quad n \geq 0 \)

Transition functions:

\( (q_0, A, \varepsilon, q_0, A) \)
\( (q_0, B, A, q_1, \varepsilon) \)

For the string ’AABB’ the actions would be:

\( (q_0, A, Z_1) = (q_0, A|Z_1) \quad \text{Tape = ABB} \)
\( (q_0, A, A) = (q_0, A|A|Z_1) \quad \text{Tape = BB} \)
\( (q_0, B, A) = (q_1, A|Z_1) \quad \text{Tape = B} \)
\( (q_1, B, A) = (q_1, Z_1) \quad \text{(End of Tape)} \)

Because we reached the end of the tape and the stack is empty(at \( Z_1 \)) the PDA accepts the string as valid.[1]

B. Context-Free Grammar

Context-Free Grammar is important for analyzing and evaluating expressions. CFG is a set of finite grammatic rules that expressions have to follow to be acceptable. Language is key to evaluating expressions correctly. To define CFG we have start with the idea of an expression which acts like a variable and has the potential to be multiple things recursively, for example an expression can be two expressions together, it can also be an operator or a verb or a function.[2]

We can think about this in a similar fashion to how spoken language is built. Lets look at a way this is done in English:

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow \text{det N} \\
NP & \rightarrow N \\
VP & \rightarrow V \\
VP & \rightarrow V \ NP \\
\text{Det} & \rightarrow (a, \text{the}) \\
N & \rightarrow (\text{cat, home}) \\
V & \rightarrow \text{runs}
\end{align*}
\]

This is an example of context free grammar, now lets apply is derive an expression:

\[
\begin{align*}
S & \rightarrow NP \ VP \\
& \rightarrow \text{det N VP}
\end{align*}
\]
C. Parsers

Parsing, refers to a process of analysing a given string and determining whether it conforms to the proper syntax and grammar. Parsing is done by two techniques. Top-down and Bottom-up, we use Pushdown Automata to recognize correct grammar, by ensuring that the PDA will only accept a string that adheres to the grammatical rules. [3]

III. Parsing Arithmetic Expressions Using PDA

We will construct a PDA capable of verifying whether an arithmetic expression is computable. Let’s begin with a description of the PDA:

\[ Q = \{ q_0, q_1, q_2, q_3 \} \]
\[ \Sigma = \{ (, ), , , +, -, *, / \} \]
\[ \Gamma = \{ (, ), \varepsilon \} \]

\( \mathbb{N} \) is all natural numbers, for the sake of brevity, when referring to operators(+,-,*,/) I will use \( \varepsilon \), as the transition actions are the same.

Transition functions:

\[ \{ q_0, (, \varepsilon, q_0, \} \]
\[ \{ q_0, , q_1, \varepsilon \} \]
\[ \{ q_0, , q_2, \} \]
\[ \{ q_1, , q_2, , q_3, \varepsilon \} \]
\[ \{ q_2, , q_0, \} \]
\[ \{ q_3, , q_0, \varepsilon \} \]

So basically the \( q_0 \) is an accepting state, allowing for new parenthesis, either a closed to end the current or an open to start a new subexpression, if at any point the wrong symbol is attempted to be popped then the PDA terminates and rejects the string, so if an expression started with a closing bracket than the PDA would terminate.

When a number is read while in \( q_0 \) then the expression moves to \( q_1 \), where it only looks for an operator, which when found moves to \( q_2 \), with this particular PDA, it’s assumed that if you want to multiply a number by the contents of a set of parenthesis, you use the multiply symbol. We push an \( \varepsilon \) onto the stack so that when we can keep track of what’s expected, so if we get a number or a parenthesis, then the \( \varepsilon \) can be popped because that’s what was expected, anything else will terminate and reject. When a parenthesis or number is found then PDA transitions back into \( q_0 \). The transition rule \( \{ q_0, , q_3, \varepsilon \} \) is for one particular situation, where we have an expression closed and that can have an operator between it and another expression, it needs to be acceptable for an operator to come between these, if in \( q_0 \) it’s not acceptable, and we can’t allow an expression to start with an operator, so this is created, where once an operator is read then it moves back to \( q_0 \).

We can see how this is handles it, like CFG, which making sure expressions follow the certain set of rules. This should accept expressions such as ‘((2+4)*3)’ and reject expressions such as ‘(2+2’ or any ill-formed expression.

Lets take the string \((3+4)\)

\[ \{ q_0, Z_1 \} \text{Tape } = (3+4)* \]
\[ \{ q_0, ]Z_1 \} \text{Tape } = 3+4* \]
\[ \{ q_1, ]Z_1 \} \text{Tape } = +4) \]
\[ \{ q_2, \varepsilon ]Z_1 \} \text{Tape } = 4) \]
\[ \{ q_0, ]Z_1 \} \text{Tape } = 1) \]
\[ \{ q_3, Z_1 \} \text{Tape } = \) \]

We reach a ‘)’ but there isn’t a rule that governs that input in the current state, so we reject this string.

Lets take a look at another string ‘(1+2)*(2+1)’

The actions would be:

\[ \{ q_0, Z_1 \} \text{Tape } = (1+2)*(2+1) \]
\[ \{ q_0, \varepsilon ]Z_1 \} \text{Tape } = 1+2)*(2+1) \]
\[ \{ q_1, \varepsilon ]Z_1 \} \text{Tape } = +2)*(2+1) \]
\[ \{ q_2, \varepsilon ]Z_1 \} \text{Tape } = 2)*(2+1) \]
\[ \{ q_0, \varepsilon ]Z_1 \} \text{Tape } = *(2+1) \]
\[ \{ q_3, Z_1 \} \text{Tape } = *(2+1) \]
\[ \{ q_0, Z_1 \} \text{Tape } = 2+1 \]
\[ \{ q_0, \varepsilon ]Z_1 \} \text{Tape } = 2+1 \]
\[ \{ q_1, \varepsilon ]Z_1 \} \text{Tape } = +1 \]
\[ \{ q_2, \varepsilon ]Z_1 \} \text{Tape } = 1 \]
\[ \{ q_0, \varepsilon ]Z_1 \} \text{Tape } = \}

The tape ends and the stack is empty, so the expression is accepted as it conforms to the syntax.

It’s important to note that PDA can not solve the actual expression, the only thing they are capable of is rejecting or accepting, they don’t return anything else and they don’t alter the tape.
IV. CONCLUSION

We have demonstrated the uses of pushdown automaton as a parser by showing the acceptance of arithmetic expressions and verifying the grammar of it through a PDA. We explained how CFGs, and parsers are used by the PDA to verify syntax, and benefits of using it. There continues to be research in this area, automata are important in the field of computation, the expansion of new ways of verifying and accepting syntax is one of the ways that this area of research can progress in.

REFERENCES

Turing Machines:
ASCII code, interpretation, encryption, and printing

J. Mason, Fictional Member, IEEE

Abstract—The purpose of this paper is to examine the connection between Turing Machines and modern computing applications such as keyboard controllers. We will construct a Turing Machine that is capable of translating binary input into ASCII output, thus exploring the principles behind both Turing Machines, input/output mechanisms, and a simplified keyboard controller process. Furthermore, we will explore possible expansions to our Turing Machine which might enhance its functionality such as the addition of a layer of encryption. Thus, this paper will include a small description of basic cryptographic principles and processes.

Keywords—ASCII, keyboard, machines, Turing.

I. INTRODUCTION

We often take for granted many processes that are critical for everyday use of personal computers. One such process is that the keyboard controller which interprets the different states of the keys, generally pressed and released, and sends relevant information about these states to the computer. At a low enough state of abstraction, this information is sent over digital circuitry as logic gates are either on or off. This binary state of electricity makes the perfect situation for a demonstration of Turing Machine functionality.

II. THE MACHINE AND ITS TAPE

A. The Tape

We will assume the machine’s tape is formatted as follows: the bulk of the tape consists of eight sequential blocks of either 1’s or 0’s each of which is separated by a single blank space represented by $B$. Of course, the ends of the tape trail off in infinite blanks. A visual demonstration of the tape follows:

... B B 0 1 0 1 0 1 0 1 B 1 ...

B. Interpreter

Next follows the machine states responsible for interpreting the binary input, but not for printing. We’ll address printing in the next section for ease of use and expansion later. For brevity’s sake, the entire table will not be constructed. Notice that the pattern remains consistent, simply creating states corresponding to binary number representation. Mark Davis’ notation will be used for ease of readability [1].

q xxxxxxx 0 R q xxxxxxx

J. Mason is a senior at Maryville College under the guidance of Robert Lowe.


C. Printing States

Next we list the aforementioned printing states. Note that not all of these characters are actually printed, but rather interpreted for formatting purposes. For simplicity’s sake we will assume that our machine is capable of printing multiple characters onto the same block if necessary. Of course, one might also desire to split the machine onto two separate tapes entirely where one machine is responsible for interpreting the binary and the other reads the result and prints the corresponding character. The printing function is mapped to the standard ASCII code chart which can be readily viewed at multiple sources including [3].

<table>
<thead>
<tr>
<th>Binary</th>
<th>Character</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>B NUL</td>
<td>0x00</td>
</tr>
<tr>
<td>00000001</td>
<td>B SOH</td>
<td>0x01</td>
</tr>
<tr>
<td>00000010</td>
<td>B STX</td>
<td>0x02</td>
</tr>
<tr>
<td>00000011</td>
<td>B ETX</td>
<td>0x03</td>
</tr>
<tr>
<td>00000100</td>
<td>B EOT</td>
<td>0x04</td>
</tr>
<tr>
<td>00000101</td>
<td>B ENQ</td>
<td>0x05</td>
</tr>
<tr>
<td>00000110</td>
<td>B ACK</td>
<td>0x06</td>
</tr>
<tr>
<td>00000111</td>
<td>B BEL</td>
<td>0x07</td>
</tr>
</tbody>
</table>

\[00010000 \ B \ BS \ qxxxxxxxx\]
\[00010001 \ B \ TAB \ qxxxxxxxx\]
\[00010010 \ B \ LF \ qxxxxxxxx\]
\[00010011 \ B \ VT \ qxxxxxxxx\]
\[00011000 \ B \ FF \ qxxxxxxxx\]
\[00011001 \ B \ CR \ qxxxxxxxx\]
\[00011010 \ B \ SO \ qxxxxxxxx\]
\[00011011 \ B \ SI \ qxxxxxxxx\]
\[00100000 \ B \ DLE \ qxxxxxxxx\]
\[00100001 \ B \ DC1 \ qxxxxxxxx\]
\[00100010 \ B \ DC2 \ qxxxxxxxx\]
\[00100011 \ B \ DC3 \ qxxxxxxxx\]
\[00100100 \ B \ DC4 \ qxxxxxxxx\]
\[00100101 \ B \ NAK \ qxxxxxxxx\]
\[00100110 \ B \ SYN \ qxxxxxxxx\]
\[00100111 \ B \ ETB \ qxxxxxxxx\]
\[00110000 \ B \ CAN \ qxxxxxxxx\]
\[00110001 \ B \ EM \ qxxxxxxxx\]
\[00110010 \ B \ SUB \ qxxxxxxxx\]
\[00110011 \ B \ ESC \ qxxxxxxxx\]
\[00110100 \ B \ FS \ qxxxxxxxx\]
\[00110101 \ B \ GS \ qxxxxxxxx\]
\[00110110 \ B \ RS \ qxxxxxxxx\]
\[00110111 \ B \ US \ qxxxxxxxx\]
\[00111000 \ B \ ! \ qxxxxxxxx\]
\[00111001 \ B \ " \ qxxxxxxxx\]
\[00111010 \ B \ # \ qxxxxxxxx\]
\[00111011 \ B \ $ \ qxxxxxxxx\]
\[00111100 \ B \ % \ qxxxxxxxx\]
\[00111101 \ B \ & \ qxxxxxxxx\]
\[00111110 \ B \ ' \ qxxxxxxxx\]
\[00111111 \ B \ ( \ qxxxxxxxx\]
\[00100000 \ B \ ) \ qxxxxxxxx\]
\[00100010 \ B \ * \ qxxxxxxxx\]
\[00100011 \ B \ + \ qxxxxxxxx\]
\[00100100 \ B \ , \ qxxxxxxxx\]
\[00100101 \ B \ - \ qxxxxxxxx\]
\[00100110 \ B \ . \ qxxxxxxxx\]
\[00100111 \ B \ / \ qxxxxxxxx\]
\[00101000 \ B \ 0 \ qxxxxxxxx\]
\[00101001 \ B \ 1 \ qxxxxxxxx\]
\[00101010 \ B \ 2 \ qxxxxxxxx\]
\[00101011 \ B \ 3 \ qxxxxxxxx\]
\[00101100 \ B \ 4 \ qxxxxxxxx\]
\[00101101 \ B \ 5 \ qxxxxxxxx\]
\[00101110 \ B \ 6 \ qxxxxxxxx\]
\[00101111 \ B \ 7 \ qxxxxxxxx\]
\[00110000 \ B \ 8 \ qxxxxxxxx\]
\[00110001 \ B \ 9 \ qxxxxxxxx\]
\[00110010 \ B \ : \ qxxxxxxxx\]
Consider the example "hello world" with an offset of +3. Thus, it's possible to implement a simple Caesar Cipher into our machine for an added level of security [2]. Encryption is. Thus, it's possible to implement a simple Caesar Cipher into our machine for an added level of security [2].

A. Caesar Cipher

One of the oldest forms of encryption is the Caesar Cipher, so named because it was supposedly used by Julius Caesar to encrypt military messages. This simple encryption technique offsets all characters within the message by a preset number. Consider the example "hello world" with an offset of +3. Thus, "hello world" becomes:

With our machine’s extra states, one could print a predefined symbol, such as gamma, to the tape before proceeding to print a message that seemingly looks like jargon. However, the party expecting the message would already know that the preceding gamma means that all following characters are off by a specific value, such as negative seven, and adjust the printed message by shifting all their values by positive seven. Of course, one could always plainly print the offset beforehand but that would make it much easier for others to guess what your method of encryption is. Thus, it’s possible to implement a simple Caesar Cipher into our machine for an added level of security [2].

D. Machine Analysis

In short our machine reads the sequential blocks of 0’s and 1’s and builds a state that corresponds to that binary number. After building that number, the machine replaces the blank dividing space it’s on with the ASCII character that corresponds to the numerical equivalent of its state and then repeats the process until terminating at the infinite blank blocks at the end of the tape.

III. MACHINE EXPANSION & ENCRYPTION

Note that our machine has states for use defined through 255, but does not use all of them for printing. Usually the remaining states would be used to map the extended ASCII character set, but there are plenty of possibilities for these unused states. For example, one might use these numbers to encrypt their binary messages. A short description of a few of the possible encryption techniques that would well with our machine follows:

A. Caesar Cipher
B. Vignere Cipher

Of course, with so many states open for use it’s possible to include more demanding encryption techniques such as the Vignere cipher. The Vignere Cipher builds upon the principle of the Caesar cipher by using several Caesar ciphers in sequence allowing for multiple offsets to exist within the same message. This sequential offset is achieved by constructing a table of the alphabet 26 times with each row shifted cyclically from the previous row, usually to the left. An example of such a table is provided below for reference:

| A B C D E F G H I J K L M N O P Q R S T U V W X Y Z |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| A B C D E F G H I J K L M N O P Q R S T U V W X Y Z |
| B C D E F G H I J K L M N O P Q R S T U V W X Y Z A |
| C D E F G H I J K L M N O P Q R S T U V W X Y Z A B |
| D E F G H I J K L M N O P Q R S T U V W X Y Z A B C |
| E F G H I J K L M N O P Q R S T U V W X Y Z A B C D |
| F G H I J K L M N O P Q R S T U V W X Y Z A B C D E |
| G H I J K L M N O P Q R S T U V W X Y Z A B C D E F |
| H I J K L M N O P Q R S T U V W X Y Z A B C D E F G |
| I J K L M N O P Q R S T U V W X Y Z A B C D E F G H |
| J K L M N O P Q R S T U V W X Y Z A B C D E F G H I |
| K L M N O P Q R S T U V W X Y Z A B C D E F G H I J |
| L M N O P Q R S T U V W X Y Z A B C D E F G H I J K |
| M N O P Q R S T U V W X Y Z A B C D E F G H I J K L |
| N O P Q R S T U V W X Y Z A B C D E F G H I J K L M |
| O P Q R S T U V W X Y Z A B C D E F G H I J K L M N |
| P Q R S T U V W X Y Z A B C D E F G H I J K L M N O |
| Q R S T U V W X Y Z A B C D E F G H I J K L M N O P |
| R S T U V W X Y Z A B C D E F G H I J K L M N O P Q |
| T U V W X Y Z A B C D E F G H I J K L M N O P Q R S |
| U V W X Y Z A B C D E F G H I J K L M N O P Q R S T |
| V W X Y Z A B C D E F G H I J K L M N O P Q R S T U |
| W X Y Z A B C D E F G H I J K L M N O P Q R S T U V |
| X Y Z A B C D E F G H I J K L M N O P Q R S T U V W |
| Z A B C D E F G H I J K L M N O P Q R S T U V W X Y |

Fig. 1. An example of a Vignere block used for the Vignere Cipher

The phrase is then encrypted using a keyword that was previously decided upon between both parties. The keyword is then repeated until it matches the length of the phrase intended for encryption. Then, starting with the first letter of the phrase, we encrypt traverse the first column until the letter is found. Then, proceed along the top rows until the character which corresponds to the same index of the keyword is found. The intersection of this row and column marks the encrypted character. Let’s consider the example with the phrase "hello world" and keyword "turing" as illustrated below:

<table>
<thead>
<tr>
<th>phrase:</th>
<th>h e l l o w o r l d</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword:</td>
<td>t u r i n g t u r i</td>
</tr>
<tr>
<td>encrypted phrase:</td>
<td>a y c t b c h l e l</td>
</tr>
</tbody>
</table>

Reversing this process with the encrypted phrase and keyword will yield the original phrase. Thus, one could modify our machine to use a Vignere Cipher using the first X number of binary strings as the keyword where X is a previously agreed upon standard. Of course, using a non-randomized keyword alongside an encrypted message would surely make it easier to crack the encryption. Thus, a randomized keyword for such a process might be preferable.

IV. Conclusion

In conclusion, this paper demonstrates a means of implementing a simple, ASCII messaging system with a Turing Machine. Additionally, we’ve demonstrated the ability to implement two simple encryption techniques into the system with the possibility to implement more advanced encryption if so desired.

ACKNOWLEDGMENT

I’m greatly indebted to Robert Lowe for his continued passion and dedication to the education of young persons such as myself. Once again I’m indebted to Maggie Mamantov who took time away from her graduate school applications and interviews to critique my grammar and style. Finally, I’d like to thank professor Lee Wittenberg for taking a genuine interest in this publication.

REFERENCES

Basic Propositional Logic Statements through Turing Machines

Matt Jenkins

Abstract—Propositional logic has important uses in computer science and mathematics, but requiring humans to work through propositional logic statements creates room for errors. Thus, we will work on creating a Turing Machine that can fulfill the role of having to calculate the answers, but it will also remove the human aspect. Because of this, it leads to answers that are much more likely to be correct as human error will have no impact on the outcome of the Turing Machine that we create so long as the statement is inputted correctly into the machine. We will be using the operations of and, or, and negation as the basis for our logic machine. After we have shown that a table for the machine can be created, we will prove each part individually. After doing that, we shall work on showing how it can also handle more general expressions that have multiple operations and blank spaces in between each value. After doing that, we will explore the idea of using the machine as a way to show equivalence between propositional statements as a major application of our machine. Finally, we will discuss improvements that can be looked at later as a way to improve the machine.

I. INTRODUCTION

Propositional logic is has a lot of uses for computer science; however, working through a statement can be much more tedious than just simply writing it down. A single human error can ruin a massive calculation; this means that if we are able to remove the human’s part of the computation, then it will probably become more consistent and error-free. Thus, our goal is to create a machine that can do this by utilizing prior work by Martin Davis and Alan Turing.

In our paper, we will only be looking at the logic operations of and, or, and negation. This will allow the machine to be useful for solving equations; however, we admit that it does have limitations as parentheses are not allowed, nor is the notation for implication allowed. Thus, there are areas of improvement that we shall discuss at a later point in the paper.

We will follow Martin Davis’s notation and description of a Turing Machine. He describes a Turing Machine as being made up of a finite, non-empty set of quadruples that instruct the machine on what they must do next [1]. This aligns nicely with a basic definition of propositional logic as there are a finite amount of decisions to be made based on the symbols (∼, ∨, and ∧) representing negation, or, and and operations, respectively. The ∨ and ∧ operations take in only two integers from the set 0, 1 and will output either a 0 or a 1 based on the inputs and which operation was chosen; for the negation operation, it will only have a single 0 or 1 as its input and output the opposite of its input. Thus, as one can see there is not an infinite amount of these combinations that can be created; therefore, we are ensured that the set of our quadruples will be completely finite and follow Davis’s format.

We shall work towards creating a Turing Machine Z and show that it can be used to compute some basic propositional logic statements. In general, this means that our machine is circular due to it writing only a single symbol on the tape (thus, this is indisputably finite) and halting [3]. We will discuss how this could be used to show equivalence between two propositional logic statements as one of the applications of our machine.

II. APPROACH

A. Language and Representation

The language of the Turing machine that we are creating will have its language represented by $S$. Thus, we let $S = \{B, 1, 0, ∨, ∧, ∼, #, %, *, @\}$. For this machine, the @ symbol will always be the right-most symbol. For us, this is utilized as a way to know that we have reached the end of the propositional logic expression on the tape. The symbols \{#, $, %, *\} are all used as a way for our Turing machine to identify which state it should be in. This means that if we encounter a $, we know that we will begin to evaluate any and expressions.

B. Turing Machine Table

The following table lists all of the quadruples that our Turing Machine knows. As the reader can see, the table is an abbreviated skeleton table that they must unpack themselves if they to see the fully expanded table. This is modeled after Alan Turing’s skeleton tables [3]; however, one should notice that we take inspiration from Martin Davis’s quadruples that consist of only one action performed per symbol scanned [1].

Inside of our table, we use the representation of the symbols instead of the $S_1$ notation. This is to help the reader, but one would be able to easily switch out every 1 for $S_1$ if they wish. By doing that, though, it creates a much more unreadable table that would require the reader to continuously look back and forth between the language’s declaration and the table. We also decided upon this way of displaying the table due to how we write each symbol in the proof that will follow the table.

Whenever one sees the notation $\sim x$ in the symbol column, this means “any symbol but x”. It must be noted, though, that if there are multiple quadruples for a single starting m-configuration, then $\sim$ only applies to symbols not present in that m-configuration’s symbol column. If we use $q_{And}$’s rows as an example, one can see that $\sim ∧$ only applies when there isn’t a $∧$, $∧$ or $@$ symbol present. Therefore, it will happen when there’s a symbol like $∨$, 1 or 0.

In addition to the $\sim$ notation, we use the phrase identifiers to mean any symbol from the set \{#, $, %, *\} ∈ S$. This is
a very minor point to be noted, but it should help the reader's comprehension of the table and how it functions.

<table>
<thead>
<tr>
<th>m-config</th>
<th>sym</th>
<th>action</th>
<th>new m-config</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>~ B</td>
<td>L</td>
<td>q1</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>#</td>
<td>q2</td>
</tr>
<tr>
<td>q2</td>
<td>~ B</td>
<td>R</td>
<td>q2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>@</td>
<td>qFront</td>
</tr>
<tr>
<td>q\text{Front}</td>
<td>~ (identifier)</td>
<td>L</td>
<td>q\text{Front}</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>$</td>
<td>q\text{Find}\neg \text{eg}</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>%</td>
<td>qAnd</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>*</td>
<td>qOr</td>
</tr>
<tr>
<td>q\text{Find}\neg \text{eg}</td>
<td>$</td>
<td>R</td>
<td>q\text{Find}\neg \text{eg}</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>R</td>
<td>q\text{Find}\neg \text{eg}</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>B</td>
<td>q\text{Neg}</td>
</tr>
<tr>
<td>q\text{Neg}</td>
<td>B</td>
<td>R</td>
<td>q\text{Neg}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>q\text{Find}\neg \text{eg}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>q\text{Find}\neg \text{eg}</td>
</tr>
<tr>
<td>q\text{And}</td>
<td>%</td>
<td>R</td>
<td>q\text{And}</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>L</td>
<td>q\text{Front}</td>
</tr>
<tr>
<td></td>
<td>~ ∧</td>
<td>R</td>
<td>q\text{And}</td>
</tr>
<tr>
<td></td>
<td>∧</td>
<td>B</td>
<td>q\text{X ∧}</td>
</tr>
<tr>
<td>q\text{X ∧}</td>
<td>B</td>
<td>L</td>
<td>q\text{X ∧}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>B</td>
<td>q\text{I ∧}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>B</td>
<td>q\text{0 ∧}</td>
</tr>
<tr>
<td>q\text{1 ∧}</td>
<td>B</td>
<td>R</td>
<td>q\text{1 ∧}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>q\text{And}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>q\text{And}</td>
</tr>
<tr>
<td>q\text{0 ∧}</td>
<td>B</td>
<td>R</td>
<td>q\text{0 ∧}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>q\text{And}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>q\text{And}</td>
</tr>
<tr>
<td>q\text{Or}</td>
<td>@</td>
<td>B</td>
<td>q\text{Fin}</td>
</tr>
<tr>
<td></td>
<td>~ ∨</td>
<td>R</td>
<td>q\text{Or}</td>
</tr>
<tr>
<td></td>
<td>∨</td>
<td>B</td>
<td>q\text{X ∨}</td>
</tr>
<tr>
<td>q\text{X ∨}</td>
<td>B</td>
<td>L</td>
<td>q\text{X ∨}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>B</td>
<td>q\text{1 ∨}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>B</td>
<td>q\text{0 ∨}</td>
</tr>
<tr>
<td>q\text{1 ∨}</td>
<td>B</td>
<td>R</td>
<td>q\text{1 ∨}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>q\text{Or}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>q\text{Or}</td>
</tr>
<tr>
<td>q\text{0 ∨}</td>
<td>B</td>
<td>R</td>
<td>q\text{0 ∨}</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>q\text{Or}</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>q\text{Or}</td>
</tr>
<tr>
<td>q\text{Fin}</td>
<td>~ *</td>
<td>L</td>
<td>q\text{Fin}</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>B</td>
<td>q\text{Done}</td>
</tr>
</tbody>
</table>

C. Proof

In order to rigorously prove that the above table works, we will show each piece individually to begin with; this will allow the reader to see exactly how each phase of the machine works in an extremely generalized way. After we have shown that each individual part works, we will show the reader in the applications section how this comes together to create a machine that can handle large, variable length expressions.

1) Negation: We shall let our tape expression be ~ V, where V represents either a 0 or 1; thus our proof is as follows.

$$\alpha = q_1 \sim V$$
$$q_1 B \sim V$$
$$q_2 \# \sim V$$
$$\vdots$$
$$\# \sim q_2 V$$
$$\# \sim V q_2 B$$
$$\# \sim V q_\text{Front}$$
$$\vdots$$
$$q_\text{Front} \# \sim V @$$
$$q_{\text{FindNeg}} B \sim V @$$
$$q_{\text{FindNeg}} B V @$$
$$B q_{\text{Neg}} B @$$
$$B q_{\text{Neg}} B V @$$
$$B V q_{\text{Front}} @$$
$$\vdots$$
$$q_{\text{Front}} B V q_{\text{Front}} @$$
$$\alpha_n = q_{\text{And}} B V @$$

The above is the full proof of negation in our machine. The value of V' at the nth step is the opposite of whatever V was at the 0th step; so, if we let V = 1, then V' = 0, and the same applies if the values are reversed. It should be noted that the we have also shown the machine transitioning from the negation state to the state where it will apply the and operation.

2) And: We will now let our tape expression be q\text{Front} % X ∧ Y @; this obviously means that we have already run through our expression searching for negations. We can let V and Y be either value from the set {0, 1}; as will be shown shortly, it doesn’t really matter to us which value is there.

$$\alpha = q_{\text{And}} % V \land Y @$$
$$q_{\text{And}} % V \land Y @$$
$$% V q_{\text{And}} \land Y @$$
$$% V q_{\text{X ∧}} B Y @$$
$$% V q_{\text{X ∧}} B Y @$$
$$% q_{\text{V ∧}} B Y @$$
$$% q_{\text{V ∧}} B Y @$$
$$% q_{\text{V ∧}} B Y @$$
$$% q_{\text{V ∧}} B Y @$$
$$% q_{\text{V ∧}} B Y @$$
$$% q_{\text{V ∧}} B Y @$$
$$% q_{\text{V ∧}} B Y @$$
$$% q_{\text{V ∧}} B Y @$$

It must first be noted that the q\text{Front} at the nth step will end up going back and beginning to scan for ∧ symbols. We said before that the values of V, Y did not matter to us and this is still true, because if one looks at the table it is based off of the value of V; therefore, on our tape when we write V ∧ Y, the Turing machine would have placed the equivalent symbol on the tape.
3) Or: Finally, we will let our tape expression be $q_{Or}V \lor Y@$. Again, this means that we have completed scanning for negation and and symbols and on our last stage. Like before, we will let $V$ and $Y$ be either value from the set $\{0, 1\}$ and it does not matter what either has as its value.

$$\alpha = q_{Or} \ast V \lor Y@$$  
$$\ast q_{Or}V \lor Y@$$  
$$Vq_{Or} \lor Y@$$  
$$Vq_{X}BY@$$  
$$qvBY@$$  
$$qvBBY@$$  
$$BqvBY@$$  
$$BBq_{v}Y@$$  
$$BBq_{Or}(V \lor Y)@$$  
$$BB(V \lor Y)q_{Or}@$$

$$\alpha_n = BB(V \lor Y)q_{Fin}B$$

This functions almost exactly like the And operation. Again, the values of $V, Y$ do not matter as the table is based off of $V$, and the symbol printed by the machine would reflect $V$'s value.

4) General Expressions: Now that we have shown each piece of the puzzle, we can now work on showing that this would work for a general, combined expression. Once this has been shown, the Turing Machine becomes much more applicable to computing propositional logic expressions.

Currently, we have proven that our machine has the ability to scan for a $\sim, \lor$ or $\land$ symbol, and once it is found handle it so long as the values it is using are located right next to the symbol. This appears to be a large limitation of our machine that impacts its usefulness, but we are able to overcome this.

In order to handle the problem, our machine must have the ability to scan left and right passing multiple blank symbols. Our m-configuration table from above handles this by splitting the action between the left and right values into two different states. So, we can work our way left until we find a value, and in addition we can do the same thing when we move scan and move right. Because we are able to skip multiple symbols, we can chain all three of the and, or, and negation functions of our machine together, because it no longer would halt if it does not immediately find a 1 or 0.

Thus, our machine then gains the ability to handle n-length propositional logic statements. We will now show a simple example of this in action for the reader. It should be clear afterwards that this can be applied to any more general propositional logical statement.

$$\alpha = q_11\land \sim 0$$  
$$q_1B1\land \sim 0$$  
$$q_2\#1\land \sim 0$$  
$$\ldots$$  
$$\#1\land q_20$$  
$$\#1\land 0q_{Front}@$$  
$$\ldots$$  
$$q_{Front}#1\land \sim 0@@$$  
$$q_{FindNeg}$1$\land \sim 0@@$$  
$$q_{FindNeg}$1$\land \sim 0@@$$  
$$\ldots$$  
$$s1 \land q_{FindNeg}0@@$$  
$$s1 \land q_{Neg}B0@@$$  
$$s1\land Bq_{Neg}0@@$$  
$$s1\land Bq_{FindNeg}1@@$$  
$$s1\land B1q_{FindNeg}@@$$  
$$s1\land Bq_{Front}1@@$$

$$\ldots$$  
$$q_{Front}s1 \land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$  
$$q_{And}s1\land B1@@$$

The above shows our machine working on computing the given tape expression. As one can see, it has the ability to skip over the blanks caused by the negation stage. Again, that ability is what allows our machine to calculate more generalized propositional statement expressions.

From this point onwards, the proof is nothing more than running through the states dealing with or and cleaning up so that we will either have all blanks or a single 1 on our tape.
Thus, the $<Res_n> = 1$ which is exactly the answer one would find if they worked this problem out by hand; therefore, we have been able to show that our machine does work in a specific case, but it is easily generalizable due to this case proving that it has the ability to chain together each operation even if there are blanks present.

III. APPLICATIONS

After showing that our machine works for general expressions, we can look at an interesting application of our Turing Machine. We can use it to show equivalence between statements. We know that we must have the same values in a truth table for two or more statements to be equivalent [2].

Thus, we can take two propositional logic statements $A$ and $B$. We can run them through our machine, setting each variable to 1 or 0 until we have exhausted each combination. After each run, we will union their two resultants like so, $R = <Res_A> \cup <Res_B>$. Thus, we will now look at $R$, using Davis’s notation. If we find that we have either two 1’s or no 1’s present on the tape, we know that this specific row of the truth table is equivalent; however, if we have a singular 1, then we know they are not equivalent.

If we are able to find that each combination of $\{0, 1\}$ for the variables in the expression is equivalent between the two, then we have found that the two expressions are equivalent to each other.

IV. CONCLUSION

We set out to create a Turing Machine that would be able to handle basic propositional logic statements. We began by creating a m-configuration table and proving that each individual operation was able to function. We then showed how we could make this applicable to all basic, general propositional logic statements. In doing so, we showcased one example of the machine working, and explained how this extended its uses to proving equivalences.

Further work on this would be to allow the machine to recognize and handle parentheses; this would allow for much more complex statements, such as showing equivalences between De Morgan’s laws.

REFERENCES

Prime Number Determination with a Multiple Tape Turing Machine using Martin Davis Notation

Michael Kaufman

ABSTRACT

Turing Machines can be used to calculate mathematical operations that could become too tedious for a human to perform. A Turing machine that calculates whether or not a given number is a prime number greater than 3 could show in a hypothetical manner how one might go about determining prime numbers in a computationally based environment. We will construct a Turing machine that calculates whether or not a given input string representing an integer greater than 3 is a prime number or not. Creating this Turing machine will functionally prove whether or not this function is computable within the confines of mechanical computation. It will be a much more simplified machine if multiple tapes are used. Therefore, we will use a multiple tape Turing machine using notation that we have created for this Turing machine implementation. Martin Davis quadruple form will be slightly modified for multiple tape usage, and I will therefore design a sextuplet form that derives from the Martin Davis form.

I. INTRODUCTION

A. Multiple Tape Turing Machine Implementation and Relevance

The algorithm on which the Turing machine runs will be reflective of a simple C++ algorithm. The algorithm involves comparing a given input to a modulus value. Instead, my algorithm will use multiplication to achieve the same affect. The first and second multiplicand begin at the value 2. The lowest possible integer tested will be 4, as that is the lowest integer that can be tested greater than 3. By exhausting all possibilities of multiplication between multiplicands until the first multiplicand reaches the input value, we will show that only when the input value is the same value as the first multiplicand, is the input a prime number [2].

This is an interesting topic regarding Turing machines especially because it has never been done before as far as can be reasonably found, and it will show how Turing machines are relatable to a programming code segment, as a function that determines factorials in this manner is easily relatable to the Turing machine implementation we will show.

B. Multiple Tape Turing Machine Notation

We will use notation for Turing machine logic similar to the logic defined by Martin Davis. However, we will use a modified version of Martin Davis notation, which is defined as using the quadruple form, as defined by the following forms:

(1) \( q_j S_j S_k q_t \)

(2) \( q_j S_j R q_t \)

(3) \( q_j S_j L q_t \)

In the first form, the first state, \( q_j \) is replaced by the state, \( q_t \), which can or cannot be the same state as \( q_j \). \( S_j \) is also replaced by symbol \( S_k \) in this same step. The second and third forms define moving right given \( R \), and moving left given \( L \) if a given symbol is on a tape with the state defined by \( q_j \). State transition into \( q_t \) can also be defined in this form [1]. The notation we have developed for a hypothetical multiple tape Turing machine is defined in a similar fashion by the following forms:

(1) \( q_j S_j S_k q_t t_i t_j \)

(2) \( q_j S_j R q_t t_i t_j \)

(3) \( q_j S_j L q_t t_i t_j \)

The notation is differentiated only by the last two symbols. \( t_i \) denotes the current tape, and \( t_j \) denotes the tape to be transitioned to. When leaving a particular tape to transition into a given state on a new tape, the current cell being examined remains the same on the previous tape when transitioned back. For our factorial determination algorithm, we will use four different tapes defined in the following forms:

(1) \( t_0 \) is defined as the input string. This string defines the number we're judging to be a prime as a sequence of 1’s.

(2) \( t_1 \) is defined as the first multiplicand string. This is a string of 1’s that will be multiplied by the second multiplicand string in order to determine if a number is not prime. This string begins as 11, as the first number to multiply will be 2. Multiplication will halt if \( t_1 \) is found to be identical as \( t_0 \).

(3) \( t_2 \) is defined as the second multiplicand string. This is similar to \( t_1 \), with the only difference being that this string is incremented first, and is not compared to the input string in determining whether to halt multiplication. A # symbol also marks the start of this string, before the 1’s.

(4) \( t_3 \) is defined as the product string. This value must be compared to the input string. If they are ever found to be identical, than the input is defined as not a prime number. This string is a # symbol followed by B symbols until 1’s are filled in to define the value of this string. Our starting position first looks at the first blank cell to the right of the # symbol.
(5) \( t_4 \) is defined as the answer string. This will be a blank cell until either a 0 or a 1 is printed, in which case a 0 denotes that the given input string is not a prime number, and 1 denotes that the given input string is a prime number.

II. APPROACH

A. The Turing Machine

\[ q_0 \ 1 \ R \ q_0 \ t_0 \ t_1 \ t_0 \] – Begin by checking to see if the input string and the first multiplicand string match, since this defines if a prime number has been found.

\[ q_0 \ 1 \ R \ q_0 \ t_0 \ t_1 \]

\[ q_0 \ 1 \ L \ q_1 \ t_0 \ t_0 \]

\[ q_0 \ R \ q_3 \ t_0 \ t_1 \] – Input string’s position has been reset

\[ q_3 \ 1 \ L \ q_3 \ t_1 \ t_1 \]

\[ q_3 \ R \ q_4 \ t_1 \ t_2 \] – Both strings have been reset in position for later comparison and such. In state 4 we begin the multiplication process.

\[ q_0 \ 0 \ L \ q_2 \ t_1 \ t_4 \] – The first multiplicand string is shorter than the input string, so we must reset to the start of the multiplicand string and the input string for later comparison.

\[ q_2 \ 1 \ L \ q_1 \ t_0 \ t_0 \] – This will go to our reset portion.

\[ q_4 \ 1 \ R \ q_4 \ t_1 \ t_1 \] – Every time we have a 1 in this multiplicand string, append the first multiplicand string onto the answer string.

\[ q_4 \ 1 \ R \ q_4 \ t_1 \ t_3 \] – We have finished multiplying. We must compare the multiplication string to the input string, and then append a 1 onto the second multiplicand string afterwards.

\[ q_4 \ 1 \ R \ q_4 \ t_1 \ t_3 \] – Put a 1 on the product tape for every 1 on our first multiplicand tape.

\[ q_4 \ 1 \ R \ q_4 \ t_1 \ t_3 \] – Once we have a blank, reset to the beginning and transition back to our second multiplicand string.

\[ q_4 \ 1 \ L \ q_4 \ t_3 \ t_3 \]

\[ q_4 \ 1 \ R \ q_4 \ t_3 \ t_3 \] – This prints 1’s as many times as we’ll need for the multiplication portion of the algorithm.

\[ q_5 \ 1 \ L \ q_5 \ t_1 \ t_1 \]

\[ q_5 \ R \ q_4 \ t_1 \ t_2 \] – This string can be iterated through again, so return to state 4 to continue multiplying.

\[ q_6 \ 1 \ L \ q_6 \ t_3 \ t_3 \]

\[ q_6 \ R \ q_8 \ t_3 \ t_3 \]

\[ q_8 \ 1 \ R \ q_8 \ t_3 \ t_3 \] – Begin comparison to input string.

\[ q_8 \ 1 \ R \ t_0 \ t_3 \]

\[ q_{13} \ B \ B \ q_{14} \ t_0 \ t_3 \] – The product was longer than the input, so we need to begin multiplying after clearing the product string and appending a 1 onto the correct multiplicand string.

\[ q_8 \ B \ q_{11} \ t_0 \ t_0 \]

\[ q_{11} \ B \ B \ q_{12} \ t_0 \ t_0 \] – The product string and the input string match, so the input string is not a prime number.

\[ q_{11} \ 1 \ 1 \ q_{13} \ t_0 \ t_0 \]

\[ q_{12} \ 0 \ q_{12} \ t_4 \ t_4 \] – The number has been determined to not be prime, and as such the Turing machine halts here with the output 0 in the answer tape.

\[ q_{14} \ B \ q_{15} \ t_3 \ t_3 \] – We are resetting for the sake of re-comparison in future runs of the machine. First, we must clear out the product string.

\[ q_{15} \ 1 \ B \ q_{15} \ t_3 \ t_3 \]

\[ q_{15} \ 1 \ L \ q_{15} \ t_3 \ t_3 \]

\[ q_{15} \ # \ q_{16} \ t_0 \ t_0 \] – We have reset the product string to look at the first blank cell to the right of the # symbol. We must now return to the beginning of the input string as well.

\[ q_{16} \ B \ q_{17} \ t_0 \ t_0 \]

\[ q_{17} \ 1 \ L \ q_{17} \ t_0 \ t_0 \]

\[ q_{17} \ B \ q_{18} \ t_0 \ t_2 \] – The input tape is looking at the first cell containing a 1. We must now decide the correct multiplicand string to increment by 1, or decide that the input is a prime.

\[ q_{18} \ 1 \ R \ q_{18} \ t_2 \ t_0 \]

\[ q_{18} \ 1 \ R \ q_{18} \ t_0 \ t_2 \] – The preceding two iterate through each other to determine equivalence.

\[ q_{18} \ B \ q_{19} \ t_2 \ t_0 \]

\[ q_{19} \ B \ B \ q_{23} \ t_0 \ t_0 \] – The strings are equivalent

\[ q_{19} \ 1 \ 1 \ q_{20} \ t_0 \ t_2 \] – The increment string is longer than the second multiplicand string, so we add a 1 to the multiplicand string, after resetting the input string’s position.

\[ q_{20} \ B \ q_{21} \ t_2 \ t_2 \] – We have appended a 1, and we must now reset the strings to look at the first element.

\[ q_{21} \ 1 \ L \ q_{21} \ t_2 \ t_2 \]

\[ q_{21} \ B \ q_{22} \ t_0 \ t_0 \] – We are at the first element of the second multiplicand string, and must now make sure we look at the first element of the input string.

\[ q_{22} \ 1 \ L \ q_{22} \ t_0 \ t_0 \]

\[ q_{22} \ B \ q_{23} \ t_0 \ t_0 \] – We return to the start of the algorithm, as we’re looking at the first cells of the applicable tapes.

\[ q_{23} \ B \ q_{24} \ t_0 \ t_0 \] – In this state, the second multiplicand has been found to be of the same length as the input.
We must restart to the beginning of input, in order to compare with the first multiplicand.

- Restart to the beginning of the first multiplicand for comparison.

We must restart to the beginning of input, in order to compare with the first multiplicand.

The strings match, so the given input is a prime number.

The input string is longer than the first multiplicand, so we must append a 1 to the first multiplicand string.

We have appended a 1, and we must now reset to the first element.

We must now set the value in the second multiplicand to 2.

The string has been cleared, so we must now set the value to 2.

We have set the second multiplicand to be equal to 2. We now can begin the iteration through the machine again, and we go to our first tape and state.

Proof

Proof by induction can show that the preceding Turing machine determines prime numbers greater than 3 accurately. A prime number is defined as any integer that can be divided only by itself and 1. Therefore, by definition of a prime number, if a number \( s \) where \( s \) is an integer, and any other number other than 1 and itself multiply to equal \( s \), then \( s \) is not a prime number.

Therefore, an algorithm can be developed that checks all possible multiplications of previous numbers up to the given input in order to check for possible multiplicands other than itself and 1. We are only testing for integers greater than 3, so the first two multiplicands are two, resulting in the answer string 4, which is then compared to the input string. If they are found to match, then the input is determined to be not prime, as multiplicands other than 1 and the input have been found. If the second multiplicand reaches the same value as the input, the second multiplicand gets set back to 2, and the first multiplicand gets incremented by 1, and is then multiplied by the second multiplicand through every incrementation of the second multiplicand until the second multiplicand reaches the input string, and then the process is repeated. When the first multiplicand reaches the input string’s value, all possible multiplicands other than 1 and the input string have been tested, and therefore it is determined that \( s \) must be prime.

This works for \( s + 1 \) as well, as the machine will simply be incrementing once more for both multiplicands, but still rigorously exhausting all possible multiplicand possibilities.

III. CONCLUSION

We have shown that a Turing machine implementation for determining if a positive integer greater than 3 is a prime number is possible. We can assume that Turing machines are much more involved than standard programming languages used today, but the possibilities that Turing machines present are certainly valid. Looking at mirroring C++ programs comparable to the algorithm that the Turing machine we developed demonstrates the length of logic Turing machines require [2]. In the future, in order to improve the machine, it might be advantageous to attempt an algorithm that can rely on a single tape. This would no doubt require an exponential increase in Martin Davis quadruples, but it would certainly demonstrate the computational possibilities of a simple single tape Turing machine. It would also be a marvel if the Turing machine could run multiple prime number computations rather than simply stating whether or not a given input is a prime number.

REFERENCES


E-Z Sequence

Kayla Brown, Maryville College, Computer Science and Mathematics Double Major

Abstract—I have derived a sequence, the E-Z Sequence, which finds the highest value of \( m \) such that \( z^m \mid n \) where \( z \geq 2 \) and \( m, n, z \in \mathbb{N} \) using the recursive sequence of

\[
an_n = \begin{cases} a_{n-1}, & \text{if } n-1 \mid a_{n-1} \\ \{n-1, a_{n-1}\}^{z-1}, & \text{otherwise} \end{cases}
\]

where \( a_1 = \{0\}^{z-1} \)

such that the value of \( m \) can be found in the sequence where \( n \) is the \( n^{th} \) position in the sequence and \( m \) is the corresponding number. In this paper, I will describe the derivation of the E-Z Sequence. In addition to this, I will also provide a proof of correctness for the E-Z Sequence and a proof that the length of the E-Z Sequence at any value of \( n \) for \( a_n \) is \( z^n - 1 \). Applications of the E-Z Sequence include to calculate primes. When the E-Z Sequence is compared to itself at every value of \( z \), we are able to find every natural divisor of every natural number. Therefore, if there exists a \( p^i \) position in the E-Z Sequence such that the corresponding number is 0 for every \( z \) except for in the case where the sequence which indicates the value of \( m \) for \( p^m \mid p \) as 1, then this number is prime because it has no natural divisors greater than 1 other than itself.

I. INTRODUCTION

In my previous paper, I suggested that a "possibility for future work is to determine primes using the Binary Carry sequence and sequences like it. The Binary Carry sequence is used to compute the value of \( m \in \mathbb{N} \) for every \( n \in \mathbb{N} \) such that \( 2^m \mid n \) where \( n \) is the \( n^{th} \) position in the sequence and \( m \) is the corresponding number. The first step to computing primes using this approach is to discover sequences that indicate the value of \( m \in \mathbb{N} \) for every \( n \in \mathbb{N} \) such that \( 3^m \mid n, 4^m \mid n, 5^m \mid n, \) and so on. The next step would be to create a Turing machine to compute the values of each position in the sequence for every sequence we have discovered. Then, for the \( i^{th} \) position in any sequence, a Turing machine exists to calculate the sequence that indicates the value of \( m \) for \( i^m \mid n \). When all the Turing machines are ran simultaneously, we are able to observe every natural divisor of every natural number. For any \( i^{th} \) position in any sequence, say we observe that every corresponding \( m \) value is 0 except for in the sequence which indicates the value of \( m \) for \( i^m \mid n \) such that this value is 1. Hence, the number \( i \) has only the divisor of itself and 1, therefore \( i \) is prime." [1]

In this paper, I will describe a sequence that indicates the highest value of \( m \in \mathbb{N} \) for every \( n \in \mathbb{N} \) such that \( z^m \mid n \) where \( z \in \mathbb{N} \) and \( z \geq 2 \). This is the first step to the process of discovering primes which I described.

A. Binary Carry Sequence

For the purpose of understanding exactly what the sequence indicating the highest value of \( m \in \mathbb{N} \) for every \( n \in \mathbb{N} \) such that \( z^m \mid n \) where \( z \in \mathbb{N} \) and \( z \geq 2 \) accomplishes, I will give a description and explanation of the sequence when \( z = 2 \). This sequence is called the Binary Carry Sequence and has been discovered to correlate with other mathematical problems. Some of these correlating problems are described in my previous paper if more information is desired.

"Let \( z \in \mathbb{N} \). Then for every natural number \( n \), there exists a natural number \( m \) such that \( z \) to the power of \( m \) evenly divides \( n \). There exists such a sequence, the Binary Carry sequence, that will indicate the value of \( m \) for every \( n \) for the value of \( z = 2 \). The Binary Carry sequence can be derived using the recursive equation \( a_n = \{a_{n-1}, n-1, a_{n-1}\} \) where \( n \in \mathbb{N} \) and \( a_1 = \{0\} \). The first few derivations of the sequence are shown in the table below." [1]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( a_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>0102010</td>
</tr>
<tr>
<td>4</td>
<td>010201030102010</td>
</tr>
<tr>
<td>5</td>
<td>0102010301020104010201030102010</td>
</tr>
</tbody>
</table>

"The value of \( m \) for every \( n \) such that \( 2^m \mid n \) can be found in the sequence where \( n \) is the \( n^{th} \) position in the sequence and \( m \) is the corresponding number. For example, to find the value of \( m \) for \( 8 \) where \( 2^m \mid 8 \), we will locate the \( 8^{th} \) position in the sequence of which the corresponding value is 3. Therefore, the highest power of 2 which evenly divides 8 is 3, hence \( 2^3 \mid 8 \)." [1]

B. Applications

Application for this sequence which has been previously mentioned is to discover prime numbers. The sequence discovers primes in such a manner that multiple sequences must be used simultaneously. The sequence described in this paper, the E-Z Sequence, indicates the highest value of \( m \in \mathbb{N} \) for every \( n \in \mathbb{N} \) such that \( z^m \mid n \) where \( z \in \mathbb{N} \) and \( z \geq 2 \). When the E-Z Sequence is compared to itself at every value of \( z \), we are able to indicate every natural divisor of every natural number [1]. Therefore, if there exists a \( p^i \) position in the E-Z Sequence such that the corresponding number is 0 for every \( z \) except for in the case where the sequence which indicates the value of \( m \) for \( p^m \mid p \) as 1, then this number is prime because it has no natural divisors greater than 1 other than itself [1].

Future research on the topic can be to find where the E-Z Sequence is applicable.

II. APPROACH

A. Notation

For the purposes of this paper, I have created a legend clarifying the meaning of each notation used. For this table, let \( z, m, n, p \in \mathbb{N} \) and \( a_{n-1} \) represent a sequence.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z^m$</td>
<td>$z$ to the exponential power of $m$</td>
</tr>
<tr>
<td>$z</td>
<td>n$</td>
</tr>
<tr>
<td>${0}^p$</td>
<td>${0, 0, \ldots, 0}$ where 0 is repeated in the sequence $p$-times</td>
</tr>
<tr>
<td>${n-1, a_{n-1}}^p$</td>
<td>${n-1, a_{n-1}, n-1, a_{n-1}, \ldots, n-1, a_{n-1}}$ where $n-1, a_{n-1}$ is repeated in the sequence $p$-times</td>
</tr>
<tr>
<td>$</td>
<td>a_n</td>
</tr>
</tbody>
</table>

This notation will be used throughout the paper, so a close study of this table will be beneficial before continuing.

**B. Determining the E-Z Sequence**

In order to find the sequence which indicates the highest value of $m \in \mathbb{N}$ for every $n \in \mathbb{N}$ such that $z^m|n$ where $z \in \mathbb{N}$ and $z \geq 2$, I started by observing the sequences for $z = 2, 3, 4, \ldots$. I created a table describing each sequence of numbers such that the $n^{th}$ position in the sequence indicates the highest value of $z$ such that $z^m|n$. Below an example of the table I created to easily observe the sequences.

| $z$ | highest value of $z$ such that $z^m|n$ for the value of the $n^{th}$ position |
|-----|--------------------------------------------------------------------------------|
| 2   | $0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, \ldots$ |
| 3   | $0, 0, 1, 0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 0, 0, 0, 1, 0, 2, 0, 1, 0, \ldots$ |
| 4   | $0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, \ldots$ |
| 5   | $0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, \ldots$ |
| 6   | $0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, \ldots$ |

When observing these sequences, I compared them all to the Binary Carry Sequence $a_n = \{a_{n-1}, n-1, a_{n-1}\}$ where $n \in \mathbb{N}$ and $a_1 = \{0\}$. I noticed that $a_1 = \{0\}^{z-1}$ for each sequence. Then, when observing how often each other number occurs, I saw that $2$ will occur between $a_2$ $z-1$ times before a different number, $2$, occurs. This sequence before $2$ occurs is denoted $a_2$. Then, $2$ will occur between $a_2$ $z-1$ times before a different number, $3$, occurs. This sequence before $3$ occurs is denoted $a_3$. This pattern continues for every number $n \in \mathbb{N}$. Using this information, I derived the following sequence, which I will refer to as the E-Z Sequence.

$$a_n = \{a_{n-1}, n-1, a_{n-1}\}^{z-1}$$

where $a_1 = \{0\}^{z-1}$

This sequence finds the highest value of $m$ such that $z^m|n$ where $z \geq 2$ and $m, n, z \in \mathbb{N}$.

**C. Examples of the E-Z Sequence**

For a thorough understanding of how the E-Z sequence operates, I will give examples of the sequence for a few small numbers. The E-Z Sequence operates the same for all values of $z \in \mathbb{N}$ such that $z \geq 2$.

For $z = 2$, the highest power of $m$ such that $2^m|n$ can be found using the sequence

$$a_n = \{a_{n-1}, n-1, a_{n-1}\}$$

such that $n$ is the $n^{th}$ position in the sequence and $m$ is the corresponding number. This is derived from the E-Z Sequence such that $\{n-1, a_{n-1}\}$ occurs $2-1 = 1$ time in the sequence $a_n$ and $\{0\}$ occurs $2-1 = 1$ time in the sequence $a_1$.

For $z = 3$, the highest power of $m$ such that $3^m|n$ can be found using the sequence

$$a_n = \{a_{n-1}, n-1, a_{n-1}, n-1, a_{n-1}\}$$

where $n \in \mathbb{N}$ and $a_1 = \{0\}$ such that $n$ is the $n^{th}$ position in the sequence and $m$ is the corresponding number. This is derived from the E-Z Sequence such that $\{n-1, a_{n-1}\}$ occurs $3-1 = 2$ times in the sequence $a_n$ and $\{0\}$ occurs $3-1 = 2$ times in the sequence $a_1$.

For $z = 5$, the highest power of $m$ such that $5^m|n$ can be found using the sequence

$$a_n = \{a_{n-1}, n-1, a_{n-1}, n-1, a_{n-1}\}$$

such that $n$ is the $n^{th}$ position in the sequence and $m$ is the corresponding number. This is derived from the E-Z Sequence such that $\{n-1, a_{n-1}\}$ occurs $5-1 = 4$ times in the sequence $a_n$ and $\{0\}$ occurs $5-1 = 4$ times in the sequence $a_1$.

For $z = 9$, the highest power of $m$ such that $9^m|n$ can be found using the sequence

$$a_n = \{a_{n-1}, n-1, a_{n-1}, n-1, a_{n-1}, a_{n-1}, n-1, a_{n-1}, n-1, a_{n-1}\}$$

such that $n$ is the $n^{th}$ position in the sequence and $m$ is the corresponding number. This is derived from the E-Z Sequence such that $\{n-1, a_{n-1}\}$ occurs $9-1 = 8$ times in the sequence $a_n$ and $\{0\}$ occurs $9-1 = 8$ times in the sequence $a_1$.

**D. Proof of Size of the E-Z Sequence**

To find the size of the E-Z Sequence at any value of $n$ for $a_n$, I calculated the size of $a_n$ at any value of $n \geq 1 \in \mathbb{N}$ for any value of $z \geq 2 \in \mathbb{N}$. I began with finding $|a_1| = \left|\{0\}^{z-1}\right|$ and the number of times that 0 appears in the sequence is $z - 1$, hence $\left|\{0\}^{z-1}\right| = z - 1$.

Then I found $|a_n| = \left|\{a_{n-1}, \{n-1, a_{n-1}\}^{z-1}\}\right|$. Since $\{n-1, a_{n-1}\}^{z-1}$ repeats $\{n-1, a_{n-1}\}$ $z-1$ times, this implies that $\left|\{n-1, a_{n-1}\}^{z-1}\right| = (n-1) + \left|\{a_{n-1}\}^{z-1}\right|$ and since $n-1$ is simply one element, $|n-1| = 1$. Hence, $\left|\{n-1, a_{n-1}\}^{z-1}\right| = (1 + |a_{n-1}|)(z-1)$. Then I calculated the value of $|a_n| = \left|\{a_{n-1}, \{n-1, a_{n-1}\}^{z-1}\}\right|$:

$$|a_n| = \left|\{a_{n-1}, \{n-1, a_{n-1}\}^{z-1}\}\right|$$

$$= |a_{n-1}| + (\{n-1, a_{n-1}\})(z-1)$$

$$= |a_{n-1}| + (1 + |a_{n-1}|)(z-1)$$

$$= |a_{n-1}| + z + z(|a_{n-1}|) - 1 - |a_{n-1}|$$

$$= z + |a_{n-1}| - 1$$

$$= z(|a_{n-1}| + 1) - 1$$
Note that
\[ |a_n| = z(|a_{n-1}| + 1) - 1 \implies |a_{n-p}| = z(|a_{n-p-1}| + 1) - 1 \]
and \[ |a_n| = |a_1| \implies p = n - 1. \] So, if \( |a_{n-p}| \) for \( 1 \geq p \geq n - 2 \) is continuously plugged into the equation derived for \( |a_n| \), the following is derived.

\[
|a_n| = z \left( \prod_{p=1}^{p=n-2} (z) \left( |a_1| + 1 \right) \right) - 1
= z \left( \prod_{p=1}^{p=n-2} (z) \left( z - 1 + 1 \right) \right) - 1
= z \left( \prod_{p=1}^{p=n-2} (z) \right) - 1
= z(z^{n-2}(z)) - 1
= z^n - 1
\]

Hence, the length of the E-Z Sequence at any value of \( n \) for \( a_n \) is \( z^n - 1 \).

**E. Proof of Correctness of the E-Z Sequence**

I will prove that the E-Z Sequence \( a_n = \{a_{n-1}, \{n-1, a_{n-1}\}^{z-1}\} \) finds the highest value of \( m \) such that \( z^m | n \) where \( z \geq 2 \) and \( m, n, z \in \mathbb{N} \) using the method such that the value of \( m \) for every \( n \) such that \( z^m | n \) can be found in the sequence where \( n \) is the \( n^{th} \) position in the sequence and \( m \) is the corresponding number.

I will first prove that \( a_1 = \{0\}^{z-1} \) holds for every value of \( z \) for every \( h^{th} \) position in the sequence where \( h \in \mathbb{N} \). All of the elements in the sequence \( a_1 \) are 0, and there are \( |a_1| = z - 1 \) elements in the sequence. Hence, for any \( 1 \leq h \leq z - 1 \), the sequence indicates that the highest power of \( z \) which evenly divides \( h \) is \( 0 \), which is clearly true.

Now I will prove that for any time when the sequence \( a_1 \) is repeated in the sequence, it will hold for every value of \( z \) for every \( h^{th} \) position in the sequence where \( h \in \mathbb{N} \). Note that the only time 0 will occur in the sequence is when it is in the sequence \( a_1 \). When \( a_1 \) is repeated in the sequence \( a_n = \{a_{n-1}, \{n-1, a_{n-1}\}^{z-1}\} \), it is always separated from the next sequence of \( a_1 \) by exactly one number, \( n-1 \). This implies that the \( h^{th} \) positions corresponding to the values where \( a_1 \) is in the sequence are in the interval \( \{xz + 1 \leq h \leq xz + z - 1\} \), which includes all \( z \)-1 positions which will correspond to the numbers in the sequence \( a_n \) only where the sequence \( a_1 \) occurs. Hence, the highest power of the \( h^{th} \) position from \( xz + 1 \leq h \leq xz + z - 1 \) such that \( z^m \mid h \) will be \( m = 0 \), which is clearly true.

Next I will prove that the sequence \( a_n = \{a_{n-1}, \{n-1, a_{n-1}\}^{z-1}\} \) will correctly indicate highest value of \( m \) such that \( z^m \mid n \) where \( n \) is the \( h^{th} \) position in the sequence. I will show this by proving that the sequence will correctly compute the desired result for any sequence beyond \( a_1 \). So, I can derive that for any \( n \in \mathbb{N} \), \( a_1 + n = \{a_n, \{n, a_n\}^{z-1}\} \). It can be seen that \( n \) will occur after any occurrence of the sequence \( a_n \) in \( a_1+n \). This implies that if \( 1 \geq x > z \) and \( x \in \mathbb{N} \), \( n \) is the \( x/(y+1) = x(z^n - 1) = xz^n \) number in the sequence. When \( n \) is the corresponding number in the sequence to position \( h \), \( h \) is at position \( x(z^n) \). Since \( x \neq z \), the highest power that evenly divides \( h \) is \( n \) hence \( z^n \mid qz^n \) which is obviously true.

Since I have proven that the sequence produced the desired results for \( a_1 \) and \( a_1+n \), I can conclude that the sequence will produce the desired result for any value of \( n \) for \( a_n \). Hence the E-Z sequence correctly finds the highest value of \( m \) such that \( z^m \mid n \) such that \( n \) is the \( n^{th} \) position in the sequence and \( m \) is the corresponding number for every sequence of \( a_n \) for any \( z \).

**III. Conclusion**

In conclusion, the E-Z Sequence finds the highest value of \( m \) such that \( z^m \mid n \) where \( z \geq 2 \) and \( m, n, z \in \mathbb{N} \) using the recursive sequence of

\[
a_n = \{a_{n-1}, \{n-1, a_{n-1}\}^{z-1}\}
\]

where \( a_1 = \{0\}^{z-1} \). Thus the value of \( m \) can be found in the sequence where \( n \) is the \( n^{th} \) position in the sequence and \( m \) is the corresponding number.

I have proven that the length of the E-Z Sequence for any value of \( z \) at \( a_n \) is \( z^n - 1 \) and that the E-Z Sequence will correctly indicate the value of \( m \) for every \( n \) under every sequence of \( z \geq 2 \).

As a result of proving that the E-Z Sequence exists and that is correct, I have also proven that there exists a sequence which exists for every natural number \( z \geq 2 \) such that for every natural number \( n \), the natural number \( m \) such that \( z \) to the power of \( m \) evenly divides \( n \) is indicated.

Future work on the topic includes implementing the E-Z Sequence to find prime numbers. Because the method had been created, the only challenge now is to find an efficient way to implement the method described in the Application section. I am confident that writing code to calculate and compare the E-Z Sequences will not be too difficult of a problem. Because the run time of implementing this algorithm is unknown at the moment, the efficiency of this problem is also unknown. Future work can also be to calculate the run time of this algorithm to finding primes and seeing if the run time is less than algorithms that currently exist for finding primes.

I am hopeful that this method will be researched more thoroughly and implemented such that more primes will be discovered.

**REFERENCES**

Computing Game Theory Outcomes with Turing Machines
Tanner Curren

Abstract—This paper aims to describe an application of Turing Machine concepts to a Game Theory problem. We briefly describe the basic ideas of game theory, after which we set up a Turing Machine that acts as two players in a Prisoner’s Dilemma game, and then makes decisions based on the best possible payoff. The application of Turing Machines to this problem provides an interesting insight into game theory problems by setting up both players as a machine, and allows us to examine games from a solely mechanized perspective. We then proceed to follow the operation of the machine on a sample Prisoner’s Dilemma game.

I. INTRODUCTION

Game theory applies mathematical analysis to competitions between more than one player in an attempt to calculate strategies for best possible outcomes for each player. We assert that we can apply the concepts of Turing Machines, which are machines composed of a set of state descriptors that move through a given input tape designed to carry out a process and reach some output, to the idea of a game’s outputs. In this paper, we will attempt to analyze a game theory problem from the perspective of Turing Machines; specifically, we will attempt to construct a game in which the players are a Turing Machine, utilize a tape that describes our outcomes, and then run this Turing Machine on a sample tape that demonstrates how the machine works. We will be utilizing a single Turing Machine for this purpose with a clear division between player states, with each state being a section of quadruples representing a single player. We will then develop the Turing Machines based on the strategies described later in the paper. We will also, to a degree, examine multiple strategies and how they interact in the game.

Constructing such a machine will allow us to examine the use of Turing Machines in a more applied context. The idea of Turing Machines working as multiple players is an interesting idea, and this paper might act as a springboard for exploring such a concept in-depth.

II. CONCEPTS OF GAME THEORY & PRISONER’S DILEMMA

We will briefly describe the concepts of game theory at a basic level, and examine a common game theory example known as the Prisoner’s Dilemma. Our Turing Machine will be run on a tape representing the Prisoner’s Dilemma.

As previously described, game theory revolves around multiple players competing for an optimal payoff in a given situation, called a game. Each player is able to make a choice out of a given set of choices, and the combination of the two choices specifies a payoff for each player. There can be any number of players in the game, and each player is able to make one choice out of any number of choices, but to keep our example simple and easy-to-understand, we will simply use two players that are each able to choose from two options. We will specify our payoffs with an ordered pair, which will first describe the payoff to Player 1, and then describe the payoff to Player 2.

In our example, we will analyze the classic Prisoner’s Dilemma. In this game, each player represents a prisoner being interrogated, and each prisoner has no contact with the other. Each prisoner can either confess to their crime or deny involvement. If both prisoners confess, they both receive 3 years in prison. However, if one prisoner denies involvement while the other confesses, then the confessor will receive no imprisonment while the denier will receive 8 years. If both prisoners deny involvement, then they both receive one year of time in prison. We can represent this game with the following table:

| Player 1 | Player 2
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Deny</td>
<td>(-1, -1)</td>
</tr>
<tr>
<td>Confess</td>
<td>(0, -8)</td>
</tr>
<tr>
<td></td>
<td>(-8, 0)</td>
</tr>
<tr>
<td></td>
<td>(-3, -3)</td>
</tr>
</tbody>
</table>

The Prisoner’s Dilemma is known as a “game of conflicting interest.” It is important to note that while this game is played without either player knowing what the other player chose, and both players choose simultaneously, other games exist in which the players take turns choosing and know each other’s choices. These games involve different outcomes depending on which player moves first, and while these present interesting scenarios, they will not be covered in this paper.

To solve the Prisoner’s Dilemma and other similar games, we will be utilizing a payoff maximization strategy; this strategy involves seeking the best possible solution for you. We will first consider the possible outcomes based on each of the opponent’s choice possibilities. Here, we see that if a player assumes the opponent chooses to confess, we have a better outcome if we also confess, and the same holds for if the opponent denies. Thus, the obvious optimal choice for this strategy is to select confess. However, we might also want to consider a situation in which there is no obvious optimal strategy; consider a variation of the game in which the double-confess and double-denial outcomes are multiplied by 10. Then, we would have a table that looks something like the following:
Then, there is no obvious solution, since if the opponent chooses deny, we would want to choose confess, while if the opponent chooses confess, we would want to choose deny. Thus, we might choose to pick whichever option gets us a higher payoff on average. Thus, choosing deny would give us an average payoff of -9, while choosing confess would give us an average payoff of -15, resulting in our choosing to deny. We could complicate this even further by introducing probabilities of the opponent’s choices, or making the game non-parallel, but for this paper we will keep our examples simple for the purpose of concept demonstration. So, our basic algorithm for such a decision might be to check if either of the opponent’s choices yields a flat-out better outcome, and if not, then average the outcomes of your decisions and choose based on that result. This algorithm will be important in building our Turing Machine.

III. BUILDING THE TURING MACHINE

Before constructing our Turing Machine that plays our game, we will first describe the tape that runs through the Turing Machine. We will be utilizing Alan Turing’s notation from On Computable Numbers, with an Application to the Entscheidungsproblem [1]. We will let the alphabet of the machine be denoted $S = \{B, ., ., |, n_0, ., n_m\}$ where the set $N = \{n_0, ., n_m\}$ represents a set of real numbers that we declare by our game’s payoff possibilities. Our tape will consist of some number in $N$ followed by a comma and another number in $N$, which represent our payoffs from the first row of the first column, then by a semicolon to separate ordered pairs. Then, the tape will contain another pair of numbers separated by a comma, representing the payoffs for row one column two, and we then separate the rows with a $|$ character. Next, we have two ordered pairs separated by a semicolon in the same fashion, representing the second row. For example, our example above would produce a tape that looks like $-1, -1; -8, 0 | 0, -8; -3, -3$ with infinite blank characters on either end of the tape.

We will then set up a corresponding set of quadruples for our machine, sectioned off by two major states, $p_1$ and $p_2$, that represent the change between players. Since our game in this sample will only have two players with two choices, we will only ever run through each player state once, but if we had larger numbers of players and choices, we would be able to simply extend the machine to accommodate the changing tape setup. Table 1 describes the quadruples that make up our Turing Machine. Note that some symbols read in will have qualifiers such as $n_3 > n_1$; these specify that the machine will move to the state if the read symbol (the symbol not already specified by the m-config’s function) meets the qualifications. We also denote repetition of operations with superscripts, where the operation is to be repeated the number of times denoted by the superscript.

<table>
<thead>
<tr>
<th>m-config</th>
<th>Symbol</th>
<th>Operations</th>
<th>m-config</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$n_1$</td>
<td>$R^3$</td>
<td>$s(n_1)$</td>
</tr>
<tr>
<td>$s(n_1)$</td>
<td>$n_2$</td>
<td>$R^3$</td>
<td>$s(n_1, n_2)$</td>
</tr>
<tr>
<td>$s(n_1, n_2)$</td>
<td>$n_2 &gt; n_1$</td>
<td>$R^3$</td>
<td>$c(n_1, n_2, n_3)$</td>
</tr>
<tr>
<td>$c(n_1, n_2, n_3)$</td>
<td>$n_4 &gt; n_3$</td>
<td>$L^{12}$</td>
<td>$r_1$</td>
</tr>
<tr>
<td>$r_1$</td>
<td>$n$</td>
<td>$(ER)^r$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>$s(n_1, n_2)$</td>
<td>$n_2 &lt; n_1$</td>
<td>$R^3$</td>
<td>$l(n_1, n_2, n_3)$</td>
</tr>
<tr>
<td>$l(n_1, n_2, n_3)$</td>
<td>$n_4 &lt; n_3$</td>
<td>$L^3$</td>
<td>$r_2$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$n$</td>
<td>$(ER)^l L^{15}$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>$s(n_1, n_2)$</td>
<td>$n_3$</td>
<td>$N$</td>
<td>$b(n_1, n_2, n_4)$</td>
</tr>
<tr>
<td>$c(n_1, n_2, n_3)$</td>
<td>$n_4$</td>
<td>$N$</td>
<td>$a(n_1, n_2, n_3, n_4)$</td>
</tr>
<tr>
<td>$l(n_1, n_2, n_3)$</td>
<td>$n_4$</td>
<td>$N$</td>
<td>$a(n_1, n_2, n_3, n_4)$</td>
</tr>
<tr>
<td>$b(n_1, n_2, n_3)$</td>
<td>$n_4$</td>
<td>$N$</td>
<td>$a(n_1, n_2, n_3, n_4)$</td>
</tr>
<tr>
<td>$a(n_1, n_2, n_3, n_4)$</td>
<td>$n_4 &gt; n_3$</td>
<td>$n_4 &gt; n_3$</td>
<td>$L$</td>
</tr>
<tr>
<td>$a(n_1, n_2, n_3, n_4)$</td>
<td>$n_4 &lt; n_3$</td>
<td>$n_3 &lt; n_4$</td>
<td>$L$</td>
</tr>
<tr>
<td>$m_2$</td>
<td>$n$</td>
<td>$R(ER)^l L^{15}$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>$m_4$</td>
<td>$n$</td>
<td>$L$</td>
<td>$m_4$</td>
</tr>
<tr>
<td>$m_2$</td>
<td>$n$</td>
<td>$L$</td>
<td>$m_2$</td>
</tr>
<tr>
<td>$m_2$</td>
<td>$B$</td>
<td>$R(ER)^l$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>$</td>
<td>$</td>
<td>$R^3$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>$n$</td>
<td>$RR$</td>
<td>$q$</td>
</tr>
<tr>
<td>$q$</td>
<td>$n$</td>
<td>$R^3$</td>
<td>$c(n)$</td>
</tr>
<tr>
<td>$c(n_1)$</td>
<td>$n_2 \geq n_1$</td>
<td>$L^4E$</td>
<td>$d$</td>
</tr>
<tr>
<td>$c(n_1)$</td>
<td>$n_2 &lt; n_1$</td>
<td>$E$</td>
<td>$d$</td>
</tr>
</tbody>
</table>

This machine should eliminate half of the remaining choices at each player’s movement, and then reach a state $d$ where the machine will halt. The remaining ordered pair should be the one chosen, and its position relative to the blanks and the $|$ which separates the choices should tell us which choice was made.

An interesting facet of the machine to note is that while it takes into consideration with each player the prediction of what the other player would be choosing, since we are simulating a game where both players choose at the same time, the machine operates by removing one set of choices at a time. Thus, we could potentially utilize this model in a way that matches the setup of a turn-based game if we altered the algorithm used to take into account the removal of the opponent’s choices. However, such an idea will not be explored in detail in this paper; we will leave this concept as a domain to be explored in future work.

IV. RUNNING THE MACHINE

As a proof of concept, we will now run our machine on our sample tape. So, we will start with the tape $-1, -1; -8, 0 | 0, -8; -3, -3$, starting at the state $p_1$, awaiting player 1’s move. We proceed through the states until we reach $s(n_1, n_2)$, at which point we can see that the left column has a better payoff than the right column in the top row by comparing $n_2$ to $n_1$. Since $n_2 > n_1$, we move to $c(n_1, n_2, n_3)$, where we see that $n_4 < n_3$. We then proceed to replace the first half of the tape with blanks, modifying our table to look like the following:
After repositioning, we then move to the state $p_2$, and player 2 then finds that there is a higher payout for choosing the bottom-right cell, where $n_2 \geq n_1$, so the machine then blanks out the first ordered pair, leaving a tape that looks like $|BBB-B -3, -3$. This is our tape’s final state, and we halt here because the machine reaches state $d$, which is terminal. Our tape then reflects the following table:

<table>
<thead>
<tr>
<th></th>
<th>Player 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deny</td>
<td>(0, -8)</td>
</tr>
<tr>
<td>Confess</td>
<td>(-3, -3)</td>
</tr>
</tbody>
</table>

Thus, we have shown that our machine works for a basic Prisoner’s Dilemma example. We can also see that the machine will also work for other examples by filling in other numbers into the tape, and the machine should operate in a similar fashion.

V. Conclusion

We have shown that we can apply the concepts of Turing Machines and computation to game theory problems, and by setting up a Turing Machine that solves a Prisoner’s Dilemma game, we have given a basic case from which we might proceed to solve other games with more in-depth machines. For example, we might expand our machine to use multiple strategies for each player, using interchangeable state descriptions for each player. We might also explore game theory-based Turing Machines that act on multiple games, generalizing our machine to recurse over multiple choices. With this basic Prisoner’s Dilemma example, we have effectively described a starting point from which we might further develop more Turing Machines for different game theory-based purposes; thus, we have established our basic set of ideas with which we might later work with in more detail.

References

Abstract—We introduce a single Turing Machine which compares two unary integers. The Turing Machine will show that the first unary integer input is greater than, less than, or equal to the second input integer. We will then prove that each function of the machine works individually using a similar method to that of Martin Davis in his book *Computability and Unsolvability* [1]. Our Turing Machine will also accept 0 as input, but will not accept negative integers.

I. INTRODUCTION

Each integer in a pair of integers is either greater than, less than, or equal to the other integer in the pair. Knowing how integers relate is important for a variety of reasons. For example, When trying to determine probabilities one might want to know how a likelihood of one occurrence happening relates to another. A comparator is also a very useful part of a computer’s architecture so creating a Turing Machine which can carry out this operation would be one way of showing that the comparison functions are computable.

In this paper we will construct a Turing Machine that compares two positive unary integers and also zero. Our machine will not include the ability to calculate negative integers. Our machine will accept input as a two unary integers or zero divided by a blank, 111B0. Upon reaching the halting stating our machine will produce a result in the form 111G11, 0L11, or 0E0.

Alan Turing in his paper *On Computable Numbers, With an Application to the Entscheidungsproblem* defines three types of machines. The type we will use in this paper is an automatic machine. These are machines completely determined by their state at each stage of motion [2]. These states will be defined with our grammar and will be used to control the motion of our machine to a halting state as used by Turing himself.

As stated, a Turing Machines grammar is how the machine runs through various states and eventually comes to a halting state assuming that the problem is computable. Our grammar will be set up as quadruples in a similar fashion to other Turing Machines. Turing introduced his grammar as quadruples which starts in some configuration, b; reads a symbol, s; performs either one single write operation, a single movement operation, or a write operation and a movement operation; and then ends in some state, c [2]. Our grammar will follow this same pattern, except we will add the constraint that each time machine takes an action it will make both a write and move operation.

Our grammar will be defined as:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin State</td>
<td>Symbol Read</td>
</tr>
<tr>
<td>$q_i$</td>
<td>S</td>
</tr>
</tbody>
</table>

where $q_i$ is the starting state, S is any symbol defined in our language below, O is the symbol to be written to the tape, M is the movement the machine takes after writing a symbol, and $q_j$ is the ending state. Note that $q_i$ can equal $q_j$. This simple grammar is all we will need to compare two unary integers as it is quite powerful. Movement operations in our machine will be limited to either left, L, or right, R, and will always come second in the operation phase of our machine.

Turing Machines usually have their own, predefined, language as well. This is a set of symbols, S, which the machine accepts as valid and can read from and write to its tape. Our language will contain the following symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>a blank</td>
</tr>
<tr>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>X</td>
<td>placeholder for 1</td>
</tr>
<tr>
<td>Y</td>
<td>placeholder for 0</td>
</tr>
<tr>
<td>G</td>
<td>greater than</td>
</tr>
<tr>
<td>L</td>
<td>less than</td>
</tr>
<tr>
<td>E</td>
<td>equal to</td>
</tr>
</tbody>
</table>

Our language is fairly compact and only contains the necessary symbols for comparing two integers. As a note, the operation state L,L would mean that the machine writes an L for less than and then moves to the left.

II. METHODS

A. The Comparator

The Turing Machine will be able to compare two unary integers as stated and return whether the first is greater than, less than, or equal to the second. The machine will consist of the quadruples:
These quadruples are semi-condensed and require the reader to do minor expansions for full apprehension. In order to prove that this grammar is correct, we will use a similar method to the one Martin Davis used in his book [1]. For greater than, we will input 11B1 which represents 2B1. It should be noted, however, that, similar to Martin Davis, our unary integers will represent integers of any arbitrary length $m_1$ and $m_2$ where $m_1$ and $m_2$ represent the first and second arbitrary unary integers, respectively. We will do the same for less than and equal to, but will use appropriately length arbitrary unary integers for these comparisons. In the case that a user enters zero on either side will will use a general method similar to the other proofs, but with a zero on one or both sides. When dealing with zero in our machine we limit the user to only input zero on a side which has zero. This means that we can safely assume that if zero is read on one side of the comparison, it is the only value on that side of the comparison. There should never be a case when the user enters zero and a unary value on the same side of the comparison and, therefore, our machine does not handle this case.

### B. Greater Than

For the proof of greater than we will set our input as two arbitrary unary integers $m_1$, represented as 11, and $m_2$, represented as 1, such that the machine starts in state $q_0 11B1$. The machine will run through the grammar and should halt in the state $q_{18} B11 G_1$ signifying that the first arbitrary unary integer is greater than the second.

<table>
<thead>
<tr>
<th>Comparator</th>
<th>S</th>
<th>O</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B q_0$</td>
<td>X,R</td>
<td>q_1</td>
<td>B,S</td>
</tr>
<tr>
<td>0</td>
<td>Y,R</td>
<td>q_0</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B,R</td>
<td>q_0</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_0$</td>
<td>X,R</td>
<td>q_1</td>
<td>B,S</td>
</tr>
<tr>
<td>0</td>
<td>Y,R</td>
<td>q_0</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B,R</td>
<td>q_0</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_1$</td>
<td>1</td>
<td>1,L</td>
<td>q_2</td>
</tr>
<tr>
<td>0</td>
<td>0,L</td>
<td>q_2</td>
<td>B,S</td>
</tr>
<tr>
<td>X</td>
<td>X,L</td>
<td>q_2</td>
<td>B,S</td>
</tr>
<tr>
<td>Y</td>
<td>Y,L</td>
<td>q_2</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B,L</td>
<td>q_3</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_2$</td>
<td>1</td>
<td>1,L</td>
<td>q_3</td>
</tr>
<tr>
<td>0</td>
<td>0,L</td>
<td>q_3</td>
<td>B,S</td>
</tr>
<tr>
<td>X</td>
<td>X,L</td>
<td>q_3</td>
<td>B,S</td>
</tr>
<tr>
<td>Y</td>
<td>Y,L</td>
<td>q_3</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B,R</td>
<td>q_4</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_3$</td>
<td>1</td>
<td>1,L</td>
<td>q_4</td>
</tr>
<tr>
<td>0</td>
<td>0,L</td>
<td>q_4</td>
<td>B,S</td>
</tr>
<tr>
<td>X</td>
<td>X,L</td>
<td>q_4</td>
<td>B,S</td>
</tr>
<tr>
<td>Y</td>
<td>Y,L</td>
<td>q_4</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B,R</td>
<td>q_5</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_4$</td>
<td>1</td>
<td>1,R</td>
<td>q_6</td>
</tr>
<tr>
<td>0</td>
<td>0,R</td>
<td>q_6</td>
<td>B,S</td>
</tr>
<tr>
<td>X</td>
<td>1,R</td>
<td>q_6</td>
<td>B,S</td>
</tr>
<tr>
<td>Y</td>
<td>0,L</td>
<td>q_15</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B.L</td>
<td>q_7</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_5$</td>
<td>X</td>
<td>X.R</td>
<td>q_5</td>
</tr>
<tr>
<td>B</td>
<td>B.R</td>
<td>q_6</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_6$</td>
<td>1</td>
<td>1,R</td>
<td>q_6</td>
</tr>
<tr>
<td>X</td>
<td>1,L</td>
<td>q_6</td>
<td>B,S</td>
</tr>
<tr>
<td>Y</td>
<td>0,L</td>
<td>q_15</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B.L</td>
<td>q_7</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_7$</td>
<td>1</td>
<td>1,L</td>
<td>q_7</td>
</tr>
<tr>
<td>B</td>
<td>G.L</td>
<td>q_18</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_8$</td>
<td>1</td>
<td>1,R</td>
<td>q_8</td>
</tr>
<tr>
<td>X</td>
<td>1,L</td>
<td>q_8</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B.L</td>
<td>q_11</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_9$</td>
<td>X</td>
<td>1,R</td>
<td>q_9</td>
</tr>
<tr>
<td>B</td>
<td>B.L</td>
<td>q_10</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_{10}$</td>
<td>1</td>
<td>1,L</td>
<td>q_10</td>
</tr>
<tr>
<td>B</td>
<td>L.L</td>
<td>q_18</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_{11}$</td>
<td>1</td>
<td>1,L</td>
<td>q_11</td>
</tr>
<tr>
<td>0</td>
<td>0,L</td>
<td>q_11</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>E.L</td>
<td>q_18</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_{12}$</td>
<td>0</td>
<td>0,R</td>
<td>q_12</td>
</tr>
<tr>
<td>B</td>
<td>B,R</td>
<td>q_13</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_{13}$</td>
<td>1</td>
<td>1,R</td>
<td>q_13</td>
</tr>
<tr>
<td>X</td>
<td>1,R</td>
<td>q_14</td>
<td>B,S</td>
</tr>
<tr>
<td>Y</td>
<td>0,L</td>
<td>q_11</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_{14}$</td>
<td>X</td>
<td>1,R</td>
<td>q_14</td>
</tr>
<tr>
<td>B</td>
<td>B.L</td>
<td>q_9</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_{15}$</td>
<td>B</td>
<td>B.L</td>
<td>q_16</td>
</tr>
<tr>
<td>$q_{16}$</td>
<td>1</td>
<td>1,L</td>
<td>q_16</td>
</tr>
<tr>
<td>X</td>
<td>1,L</td>
<td>q_16</td>
<td>B,S</td>
</tr>
<tr>
<td>B</td>
<td>B,R</td>
<td>q_17</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_{17}$</td>
<td>1</td>
<td>1,R</td>
<td>q_16</td>
</tr>
<tr>
<td>B</td>
<td>G.L</td>
<td>q_18</td>
<td>B,S</td>
</tr>
<tr>
<td>$q_{18}$</td>
<td>1</td>
<td>1,L</td>
<td>q_18</td>
</tr>
<tr>
<td>0</td>
<td>0,L</td>
<td>q_18</td>
<td>B,S</td>
</tr>
</tbody>
</table>
Which is a halting state. Thus the machine works for any two unary integers for which the first integer is greater than the second.

C. Less Than

For the proof of less than we will set our input as two arbitrary unary integers $m_1$, represented as 1, and $m_2$, represented as 11, such that the machine starts in state $q_{01B11}$. The machine will run through the grammar and should halt in the state $q_{18B111}$ signifying that the first arbitrary unary integer is less than the second.

Which is a halting state. Thus the machine works for any two unary integers for which the first integer is equal the second.

E. General Proof With Zero

For the proofs involving zero, we first reassert that if the input has a zero on either side of the comparison it must be the only value on that side in order to be valid. To prove that our Turing Machine correctly handles zero in all three cases, greater than, less than, and equal to, we will set up a proof for each one in similar format to the proofs we have completed above.

1) Greater than with Zero: For the case that the user inputs zero as the second integer and any integer greater than zero as the first, we will set our input so that $m_1$ is any arbitrary unary integer, represented as 1, and $m_2$ is zero, represented as 0, such that the machine starts in state $q_{01B0}$. The machine will run through the grammar and should halt in the state $q_{18B1G0}$, signifying that the first integer, which could be any random unary integer, is greater than zero. The machine runs as follows:

Which is a halting state. Thus the machine works for any two unary integers for which the first integer is less than the second.

D. Equal To

For the proof of equal to we will set our input as two arbitrary unary integers $m_1$, represented as 1, and $m_2$, represented as 1, such that the machine starts in state $q_{01B1}$. The machine will run through the grammar and should halt in the state $q_{18B1E1}$ signifying that the first arbitrary unary integer is equal to the second.

Which is a halting state. Thus the machine works for any two unary integers for which the first integer is less than the second.
Which is a halting state. Thus, our Turing Machine correctly computes comparisons with any arbitrary unary integer first and zero second.

2) *Less than with Zero:* For the case that the user inputs zero as the first integer and any integer greater than zero as the second, we will set our input so that \( m_1 \) is zero, represented as 0, and \( m_2 \) is any arbitrary unary integer, represented as 1, such that the machine starts in state \( q_00B1 \). The machine will run through the grammar and should halt in the state \( q_{18}B0L1 \), signifying that the first integer, zero, is less than the second. The machine runs as follows:

\[
\begin{align*}
Bq_00B1B \\
BYq_0B1B \\
BYBq_1XB \\
BYq_2BXB \\
Bq_3YBXB \\
q_3Bq_1XB \\
Bq_2YXB \\
B0q_12BXB \\
B0Bq_13XB \\
B0B1q_14B \\
B0Bq_91B \\
B0q_9B1B \\
Bq_180L1B \\
q_{18}B0L1B
\end{align*}
\]

Which is a halting state. Thus, our Turing Machine correctly computes comparisons with zero first and any arbitrary unary integer second.

3) *Equal to with Zero:* For the case that the user inputs zero as the first and second integers, we will set our input so that \( m_1 \) is zero, represented as 0, and \( m_2 \) is also zero, represented as 0, such that the machine starts in state \( q_00B0 \). The machine will run through the grammar and should halt in the state \( q_{18}B0E0 \), signifying that zero is equal to zero. The machine runs as follows:

\[
\begin{align*}
Bq_00B0B \\
BYq_0B0B \\
BYBq_10B \\
BYq_2BYB \\
Bq_3BYYB \\
q_3Bq_1YBB \\
Bq_2YXB \\
B0q_12YBB \\
B0Bq_13YB \\
B0q_13B0B \\
Bq_180E0B \\
q_{18}B0E0B
\end{align*}
\]

Which is a halting state. Thus, our Turing Machine correctly computes comparisons between two zero values.

### III. Conclusion

In summary, we have shown that creating a Turing Machine which compares two unary numbers is possible and, therefore, is a computable problem. Our machine took in two unary number or zero separated by a blank, 111B11, 0B111, or 0B0, and produced a result in a similar form showing that the first number was greater than, less than, or equal to the second number. To continue this work you could add another Turing Machine which uses this one to compare two values in order to help interpret data, or just simply add the ability to compare negative numbers.

### References
