Syrian Arab Republic
Ministry of Education
National Center for the Distinguished

A Comparison Between Two Models of Computation

A seminar in mathematics

By Khaled Ismaeel
Supervised by Mr. Habeed Issa

12/7/2015

Contents

1. Preface

Ever since ancient times, mankind have adapted numerous ways to do their computation. Everything started out when ancient sheep keepers wanted to ensure that at the end of a long workday, no sheep were lost. And they've done that by counting the sheep using stones. And arithmetic was born. As the human race developed, so did its need of computation. And the formulas they used in order to express the world around them got more and more complicated. Until the point where they started building special purpose machines to solve these computational problems around them. And the abstract concept these machines were built on is what's known as the "Computational Model" or "Model of Computation".

Early models of computation include Babbage's "Difference Engine". Which was a mechanical device capable of basic arithmetic operations, such as addition and subtraction. Although addition is a relatively easy task for the human mind compared to, say, solving third-order differential equations, but the difference engine was very, very fast. Simply because the position of a node in the Difference Engine, unlike a thought cannot be misinterpreted. After that, the second generation of the Difference Engine came about. And this one was capable of solving ballistics equations. Now, we're going somewhere.

And of course, how can we forget about the famous Enigma code! and the genius who cracked it, Alan Turing. What really happened is that Turing discovered the flaw in the Enigma encryption scheme. But the heavy duty work was left for Bombe, a powerful computer that Turing built in order to cope with the enormous computations involved in cracking the code.

In the present, different discoveries led to different models of computation. Such as the Turing machine, cellular automaton and others. Although all of them have been proven to be equivalent, some are better at certain tasks than others. furthermore, some gave birth to entirely new scientific concepts. This does beg the question; why do we use so many models of computation? What advances did this diversity lead to? And most of all, is there some superior model that can be used instead of all the others combined?

In this seminar, we shall introduce some of the most famous models of computation. Later, we will look at different comparisons between these models, to show the strengths and weaknesses of each model. Hopefully, we will be able to know when to use each model, and therefore choose the optimal model for several scientific fields.

2. The models

First, we will introduce the two models of this research. And we start we the most famous model of them all, the Turing machine.

1.1. The Turing machine

The Turing machine is a model first proposed by the English Scientist Alan Turing. Its basic structure is in general much older. Basically, you write the input on a tape, and get a result out on another tape by applying certain physical rules. And in the process, you write down intermediate results on scratch tape, much like a draft paper.

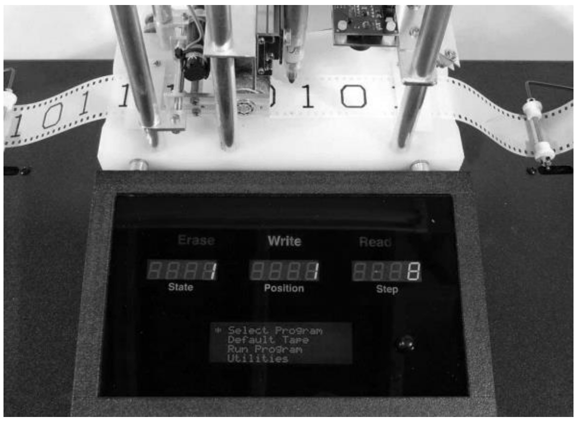

Figure 1: A mechanical Turing machine. Even though Turing machines are not necessarily mechanical devices, it's nice to build such a replica.

The Basic Structure

The K-tape Turing machine consists of the following components:

- **A scratch pad**: the scratch pad is a set of K tapes. Each tape is an infinite one direction vector of cells. Each cell contains a symbol from a fixed set of symbols R called the language.
  Each tape is equipped with a tape head. The tape head is able to either read symbols from a single cell or write a symbol on it. And the machine's computation is divided into discrete time units called steps. With each step, the head either read/writes a symbol, or moves left or right by one cell, or the machine halts. The mechanism for choosing which operation to apply will be discussed later.
  The first tape of the machine is called the input tape, and its head is only capable of reading symbols, not writing them.
  The rest of the k-1 tapes are called work tapes. And the last one is designated as the output tape, where the machine writes down the output of the program before it halts.
- **A finite set of rules**: the machine has a finite set Q of states. And it contains a register that stores one element of Q, that is the state of the machine at that instant. This state is what determines the action at the next computational step, which consists of the following: 1. Read the symbols under the k heads. 2. For the k-1 work tapes, replace

each symbol with a new one (which could be the same as the old symbol) 3. Change the value the register is storing to contain a new state from the set Q. 4. Move each head one cell to the left or right, or stay in the current position.

Formally speaking, we give the following definition:

The Turing machine M is a triple (Γ, Q, δ) where:

- Γ is a finite set of symbols. Knowing that is contains two special symbols, the start and blank symbols, denoted $' \rhd '$ and '$\blacksquare'$ respectively. The set Γ is called the alphabet of the Turing machine M.
- Q is a finite set of states that M's register can be in. again, this set contains two special element, called the start and halt states, denoted q_{start} and q_{halt} respectively.
- $\delta: Q \times \Gamma^k \longrightarrow Q \times \Gamma^{k-1} \times \{L, R, S\}^{k-1}$ is a function that describes the rules that M follows during its execution, called the transition function.

Each tape's cells, are initialized with the start symbol $' \rhd '$ at the leftmost position, and with the blank symbol $'\blacksquare'$ in every other cell. While the input tape is initialized with the start symbol at the left most position, plus a non-empty string of symbols which in the input, and the blank symbol in every other cell. The tape heads position is initialized to the beginning of the tape, where the start symbol is. In these initial conditions, the machine is in the start state q_{start}.

The transition function works as follows: let the machine be in state q and the symbols read under the k read heads are $s_1, s_2, \dots, s_k$, respectively. And: $\delta\big(q, (s_1, s_2, \dots, s_k)\big) = (q', (s_2', s_3', \dots, s_k'), z)$ where $z \in \{L, R, S\}^{k-1}$. This means that the write heads will replace each symbol s_i with symbol s_i' for $i \in \{2, \dots, k\}$. And the machine will be in state q'. After that each head will move either left or right or stay in place, as described in z where L denotes moving left, R denotes moving right and S denotes staying in place.

Each step of computation is performed by applying the transition function as described above. When the machine is at the state q_{halt} the transition function stops and doesn't make any further modification to the scratch pad.

Note that at first glance, the Turing machine seems like a mechanical or electrical device. But, in fact, the Turing machine is not restricted to certain hardware. It is rather an abstract mathematical concept. And throughout this seminar, we will treat it as a mathematical structure.

Here's a good example. Let PAL be the function defined by:

$$PAL : \{0,1\}^* \longmapsto \{0,1\} : PAL(x) = \begin{cases} 1: x \text{ } is \text{ } a \text{ } palindrome \\ 0: otherwise \end{cases}$$

Right now we will show a Turing machine Mthat computes the function PAL.

Our machine will use the language $\Gamma = \{\rhd, \blacksquare, 0, 1\}$ and three tapes. An input tape, an output tape and a work tape. And it works as follows[1]:

- Copy the input to the work tape.
- Move the input head to the beginning of the tape.
- Move the input head to the right while moving the work head to the left. If the heads read two different symbols, halt and return 0.
- Return 1.

It's not precisely the correct way to describe a transition function, but we will not introduce the full way. To see the full description, see [1]

Unnecessary modifications

Over the years, several modifications to the Turing machine have been proposed in order to enhance the capabilities of the machine. But, as it turns out, these modifications don't add any new abilities to the model. But rather affects the running time of the machine. Here are some of the modification.

- **Expand the language Γ to have more and more symbols**: Let M be a Turing machine that uses a language Γ. Here, we can map the language Γ to a language Γ' of $|\Gamma|$ strings, where each of these strings is $\log|\Gamma|$ bits long. So we can up with a Turing machine M' that uses a the language $\{\rhd, \blacksquare, 0, 1\}$ to do exactly the same job. It can be proven that for any Turing machine M that uses language Γ and any *Time-Constructible* T, if M computes x in $T(n)$ steps, then one can use TM M' that uses the language $\{\rhd, \blacksquare, 0, 1\}$ to compute x in at most $4\log|\Gamma|T(n)$ steps.
- **Increase the number of work tapes of the machine**: Let M be a TM that uses k work tapes. In fact, we can represent the strings on these k work tapes as a single string on one tape as follows. We put the first bit of the first tape in the first cell, the second bit in the $k + 1^{st}$ cell, and so on. The first bit of the second tape in the second cell, the second bit in the $k + 2^{nd}$ and so on. The l^{th} bit of the i^{th} tape in the $l + ik^{th}$ cell. So we can up with a TM M' that uses only three tapes that does the same job as M. It can be proven that for any TM M that uses k tapes, and any *Time-Constructible* T. If M computes x in $T(n)$ steps, then one use TM M' that uses 3 tapes to compute x in at most $5kT^2(n)$ steps.

[1] Arora, S., & Boaz, B. (2009). *Computational complexity: a modern approach.* Cambridge University Press.

- **Using bidirectional tapes**: Let M be a TM that uses bidirectional tapes. Obviously, we can represent each bidirectional tape as two tapes, one that stores cells $0, 1, 2, \dots$ and another that stores cells $-1, -2, \dots$. It can be proven that for any TM M that that uses bidirectional tapes and any *Time-Constructible* T. If M computes x in $T(n)$ steps, then one can find an ordinary TM that computes x in $4T(n)$.

Other modifications include using 2- or 3- dimensional tapes, allowing the machine to random-access its tapes and making the output tape write-only.

The universal Turing machine

First, we have the following observation. We can express every Turing machine M as a finite string. Indeed, since M's language and states and transition can be written down, they can encoded in binary. Furthermore, every string represents *some* Turing machine. If M is a TM, we denote by $\underline{M}$ M's binary description. And if α is some string, we denote by M_α the TM that is expressed as α.

It was Turing that first observed that general purpose computers are possible, by showing a universal Turing machine that can simulate the execution of every other TM M given M's description as input. More formally, we have the following theorem:

Theorem: **The efficient universal Turing machine**[2]:
There exists TM U such that for every $x, \alpha \in \{0,1\}^*, U(x, \alpha) = M_\alpha(x)$. Furthermore, if $M_\alpha(x)$ halts within T steps , then $U(x, \alpha)$ hats within $CT\ log\ T$ steps, where C is a constant that depends on M 's language size, number of tapes and number of states.

The above efficient construction was due to Hennie and Stearns [HS]. Right now we will show a universal TM U that is less efficient, as it computes the $M_\alpha(x)$ with quadratic slowdown. Hennie and Stearns construction can be found in [2].

Based on the above claims, we shall reduce the simulated machine to the machine M that uses the language $\{\rhd, \blacksquare, 0, 1\}$ and three work tapes. Our universal TM U will use the language $\{\rhd, \blacksquare, 0, 1\}$ and two input and output tapes, with an additional three work tapes. One tape stores the description of M's transition function, which is represented as a table of pairs $(x, \delta(x))$. Another stores M's current state. And the last stores a simulation of M's work tapes. To simulate one step of M's execution, U scans the transition function table and current state, to find out the

[2] Arora, S., & Boaz, B. (2009). *Computational complexity: a modern approach.* Cambridge University Press.

new state, symbols to be written and head movements. It is obvious that for each step of M, U executes C steps, where C is a constant depending of the size of M's transition function.

The non-deterministic Turing machine

The concept that we have been talking about is called "deterministic Turing machine". The word "deterministic" means that the machine can go from one state to one and only one other state. The non-deterministic Turing machine is a generalization of the Turing machine, so that it can go into multiple states from a given state. Formally speaking, the non-deterministic Turing machine is a generalization so that the transition mapping is not necessarily a function, i.e. from a given state the transition can compute more than one transition commands, meaning the machine can go through more than one decision paths from a given state. If we think of the decisions that a Turing machine encounters as a tree, the deterministic Turing machine goes through on path from the root to one leaf, while the non-deterministic Turing machine goes through the entire graph in a Breadth-First-Search.

This model is in fact a purely theoretical model, since that no non-deterministic Turing machine has been built yet. But as we shall see later, this concept has had a stunning effect on the complexity theory.

1.2. Cellular automaton

Cellular automaton is another discrete model of computation first suggested by John von Neumann and Stanislaw Ulam in the Los Alamos national Laboratory. Later on, Stephen Wolfram worked on the subject and published in 2003 the magnificent *A New Kind of Science*. This model has also been shown to be universal, i.e. it is able to simulate every other model of computation, including the Turing machine.

Structure

A cellular automaton simulation consists of a regular grid of cells of any finite number of dimensions. Each of these cells has a finite number of states, much like symbols on the Turing machine tape. Then we assign each cell to another finite set of cells, called its neighborhood. The simulation is divided into discrete time steps, called generations. The rule is what determines the state of the simulation. The state of each cell in the $n + 1^{st}$ generation depends on its state and the state of its neighborhood in the n^{th} generation. The state which the simulation started in is called the initial state.

Different kinds of "neighborhoods" have been proposed over the years. The most famous ones are the Von Neumann neighborhood and the Moore neighborhood. As shown in the figure. Another one is the extended Von Neumann neighborhood. This is part of the beauty of cellular automaton, the unlimited possibilities. But, in this seminar, we will study only few of these possibilities. For further reading, see [7].

Examples

Let's take a look at an example, it is called *Conway's game of life*, due to the English mathematician John Conway. The simulation consists of a regular infinite two-dimensional grid. Each cell has two states. On, which he called alive. And off which he called dead. The neighborhood of each cell is the 8 adjacent cells. There is a simple set of rules to determine each cell's state at the following generation, as follows:

1- If an alive cell has 1 or less alive neighbor cells, it becomes dead in the next generation. (as if due to loneliness).
2- If a dead cell is surrounded by 3 alive cells, it becomes alive in the next generation. (as if by reproduction).
3- If an alive cell is surrounded by 4 or more alive cells, it becomes dead in the next generation. (as if due to overpopulation).

So if we were to run the simulation on the initial state as shown in the figure below, we would have a series of generations as shown.

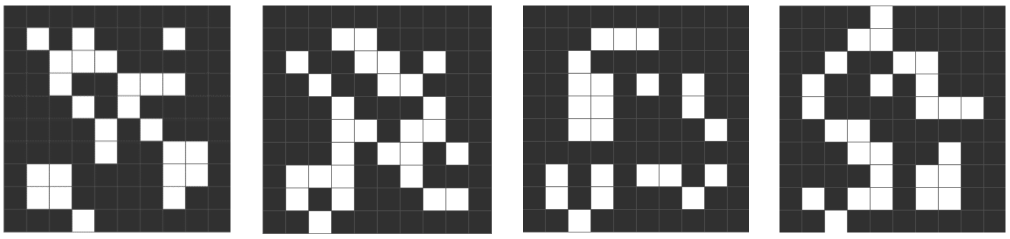

Figure 3: A 5-generation simulation of Conway's game of life using Golly 2.5 software. The initial pattern (left) yields the second pattern after applying the rules as described. The second pattern yields the third, and so on.

Another famous example is the automaton used by Stephen Wolfram, which he calls "elementary automaton". This is a one dimensional automaton. and the state of each cell in the next generation depends on its state and the state of the two neighboring cells in the current generation. There are only $2^3 = 8$ possible configurations of a neighborhood, so we can enumerate the neighborhoods from 0 to 7 based on their shape. Then we can represent a rule using a string of 8 bits, with the i^{th} bit referring to the state of cell in the next generation if it has the i^{th} neighborhood, this is the Wolfram notation. As shown in figure 3 below.

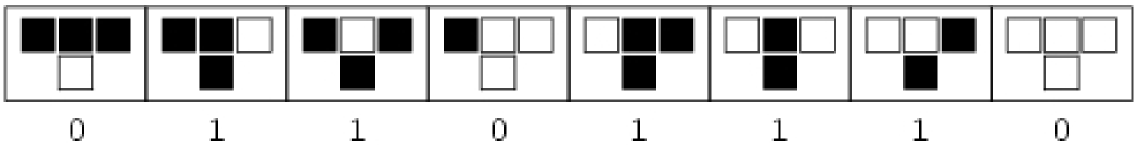

Figure 4: A representation of rule 110 using wolfram notation. Each block represents a neighborhood and the state of the center cell in the next generation. The string below the blocks is the representation. It is obvious that $01101110_2 = 110$*, which is the way rules are enumerated.*

One beautiful aspect of the elementary automaton is the ability to represent the simulation as a function of time. since it is only one dimensional, one can use a two axis, one for the simulation at a specific generation, and another to simulate the generation. As shown in figures 4 and 5.

These are only a few examples of how cellular automaton can be used. Later we will see how useful such simulations can be.

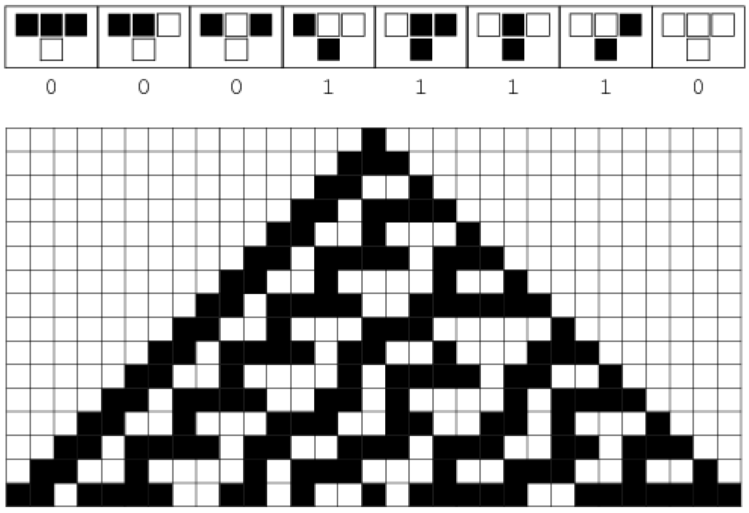

Figure 4: (above) A representation of rule 30 using Wolfram notation. (below) A simulation of rule 30 for 15 generations. Each generation is one line. For each generation, the new generation is directly below it.

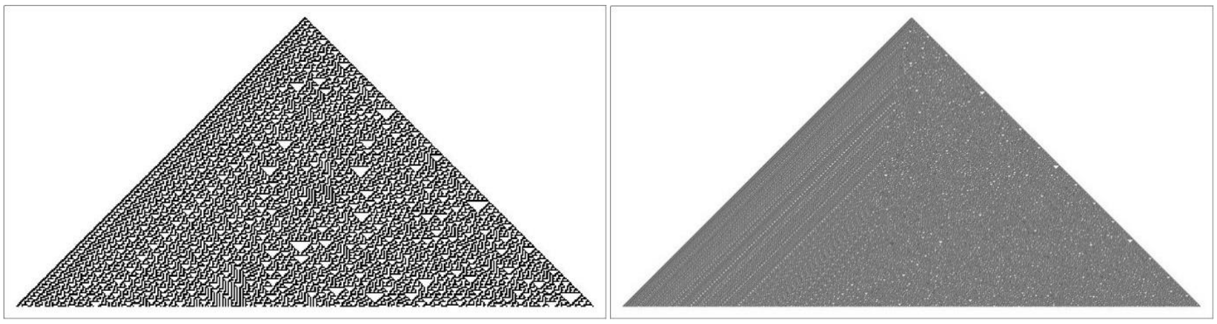

Figure 5: The simulation of rule 30 for 200 generations (left) and 1000 generations (right) made using Wolfram Mathematica 10.2 software. As we can see, in the left side of simulations there is pattern. However, this not the case in the right side, this is why rule 30 is particularly interesting, because almost all of the 256 rules don't have this chaotic behavior.

Other Variations

What we have presented above are some simple cellular automaton rules. There are several features of cellular automaton that go beyond the previous examples. Take for example the hexagonal automaton, a cellular automaton with a grid of regular hexagons. Or the continuous cellular automaton, where the set is a continuous set, and the rules are continuous functions over that set. There is also the reversible cellular automaton, which has rules in such a way that they are reversible. There are lots more, but they will remain outside the scope of this research.

3. Comparison

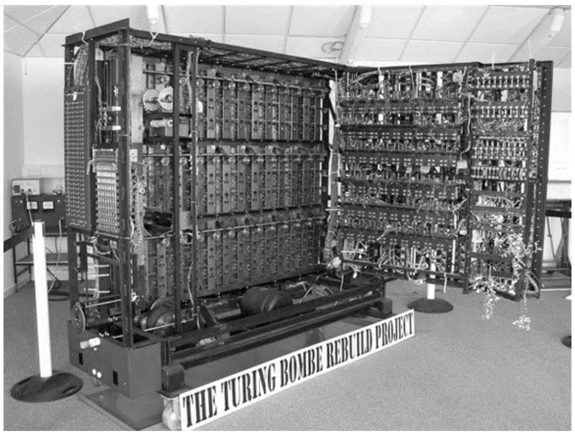

Figure 6: A working replica of the Turing machine that Alan Turing built in the 1940's, called Bombe. This replica has been built using complex electric circuits.

Right now we will make a simple comparison. We will not examine topics in detail, but rather just take a high level view on the differences between the models.

It has been proven has these two models of computation are both universal. i.e. they are both able to simulate every other model. However, we will see that the Turing machine is more adequate for some applications than cellular automaton and vice versa. Furthermore, there are some breakthroughs in mathematics and theoretical computer science that have been made using a model that would otherwise be almost impossible to reach using other models.

The Turing machine is in fact the most famous model of computation, and model-of-choice for most of the mathematicians, complexity theorists and computer scientists. Due to its easy-to-understand structure, ideal for university students. And its simple structure made it easy and fast to interpret, besides being easy to express in mathematical formulas. without this beautiful simplicity, most of the breakthroughs in complexity theory in the last 40 years would have been insanely difficult, if even possible.

Thanks to the universal Turing machine, we can now build general purpose computers. In fact, all the computers we have are built based on the universal Turing machine design. Compiling a certain code is in fact the process of transforming the code into a description of a Turing machine that executes that algorithm. The reason behind this is that the Turing machine is a more physically realizable model than cellular automaton, since it can be interpreted easily using electric circuits. Cellular automaton, however, is not that simple to use, because the simulation of its rules is not very efficient to interpret. And it rather stays a more theoretical model. When Alan Turing broke the Enigma code, he used a Turing machine called colossus, which he built using electrical circuits and valves. Trying to achieve that goal with cellular automaton would have been almost impossible.

This does beg the question, why cellular automaton? Well, cellular automaton was first invented when scientists tried to simulate natural phenomena with a model so that they can use the model to predict the future behavior of the object/system. This is still the case these days. Different cellular automaton rules are being used. For example, Wolfram has used cellular automaton to simulate a fluid dynamical system. The way he did this, he set every particle as a cell, and derived the rule from the forces that govern the system at the atomic level. But in order to get results in a scale realizable to human senses, he zoomed out and set every cell as the average of the states of a wide neighborhood of cells. As shown in figures 7 and 8.

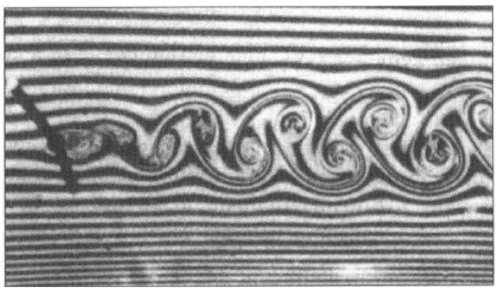

Figure 7: A vortex disturbed by an obstacle, taken from Wolfram's book. Wolfram later used cellular automaton to simulate this system.

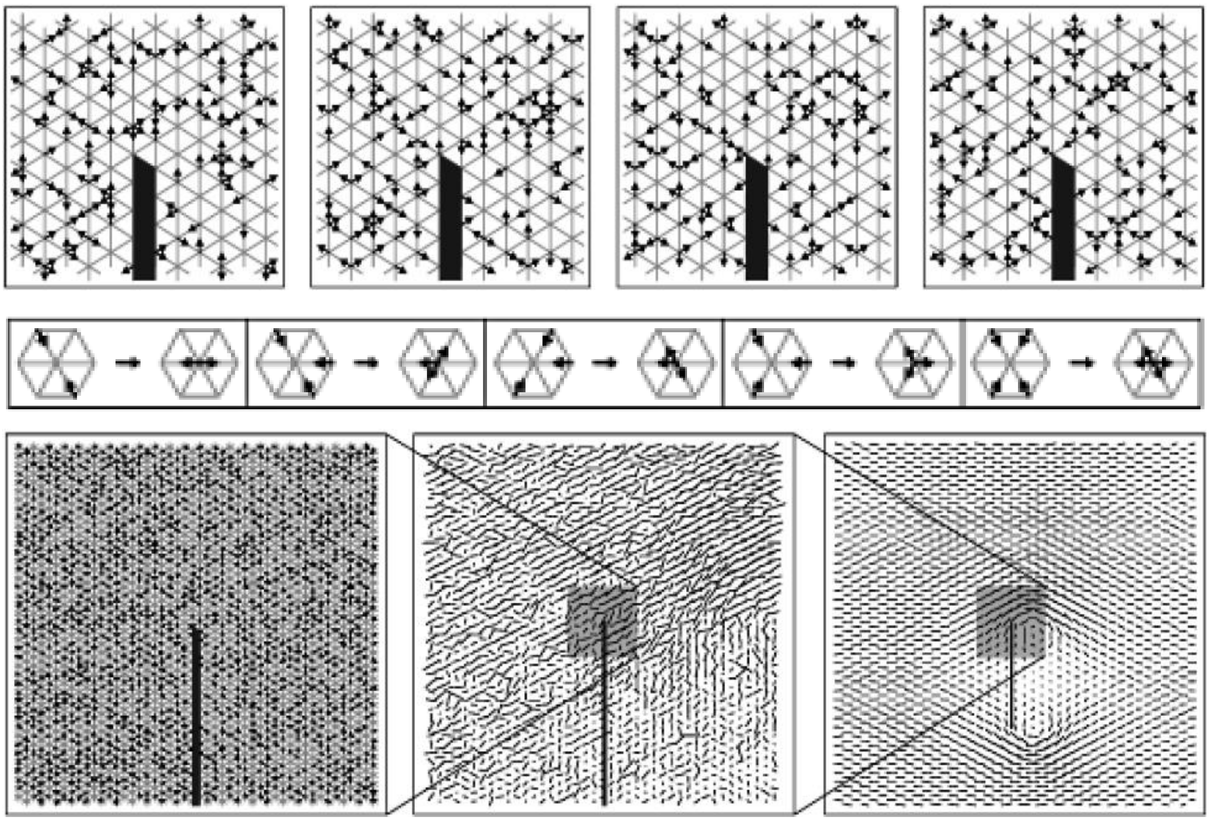

Figure 8: The cellular automaton used by Wolfram to model the vortex shown in figure 7 taken from his book. The rule for this automaton, as shown in the middle, is derived from the forces that govern this system at the atomic level. The automaton scaled out so that each vector (cell) represents the average of the 25x25 neighbor vectors, as shown in the bottom.

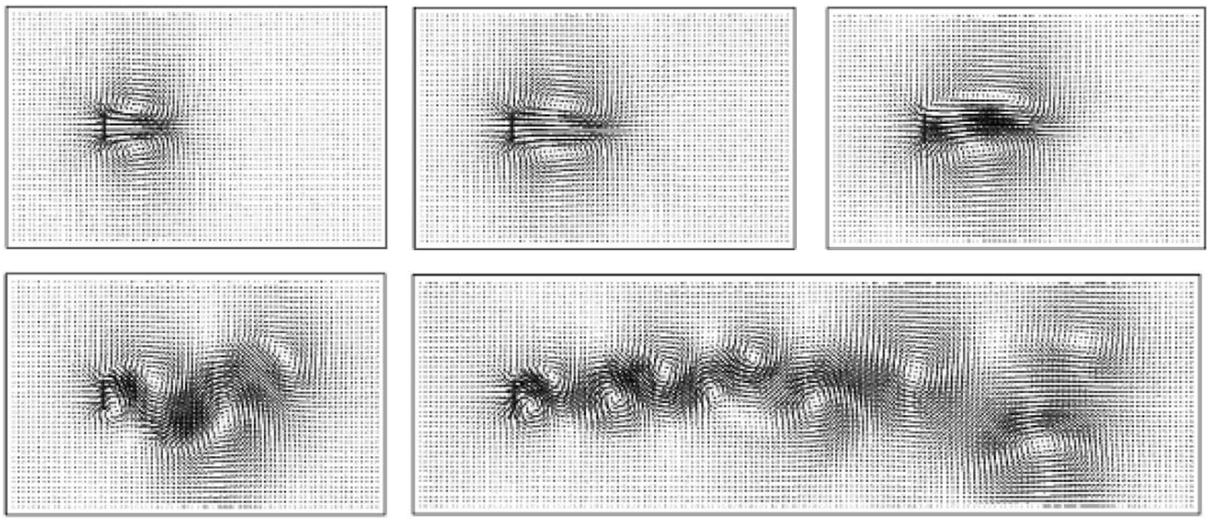
Figure 9: The evolution of the automaton as mentioned above in figure 8, in generations 10000, 20000, 30000, 40000 and 70000 respectively. As we can see the simulation comes closer and closer to the shape of the vortex in figure 7.

Other fields where cellular automaton has been used by scientists are:

- Traffic modeling: One can use cellular automaton –elementary and multi-dimensional– to simulate the traffic behavior in roads and junctions, and to predict the future state of the traffic in order to avoid potential traffic jams. This could solve lots of problems at crowded cities such as London.
- Structural design: When one is working on a design, he is restricted to several constraints, such as weight, strength, cost, etc. Cellular automaton has been used to check the designs for capability, and even for creating new designs.
- Biological systems: After observing and studying certain species for a long period of time, one can come with a rule that simulates the behavior of these species. John Conway has created the game life based on a high level view of this subject.

These are only a few of the many applications that cellular automaton can be used for. Trying to achieve the above goals with a Turing machine can be very painful when converting the objects into bit strings and the rules into transition functions. See Further reading for more information about the subject.

Let's now talk about the theoretical impact of these two models.

Over the past 40 years, complexity theory has evolved a lot. The result was dividing the computational problems into complexity classes, based on space and time complexity. But, based on what do you determine the space and time complexities? Well, the complexity theorists have used the Turing machine as the model of choice for most of their studies. And it's not surprising, because it's much more obvious to transform an algorithm to a description of some Turing machine than to transfer it to the rule of some cellular automaton.

Even though the non-deterministic Turing machine has not been built yet, the concept of non-determinism has had great significance on complexity theory. Many studies of the non-deterministic complexity classes have yielded great results concerning deterministic complexity classes. Such as Savitch's theorems [3] and the hierarchy theorems. Furthermore, most of today's asymmetric encryption schemes are based on $NP - Complete$ problems, which is a complexity class defined based on the non-deterministic Turing machine.

Here are some of the complexity classes that complexity theorists have developed over the years:

Complexity class	Description
P $DTIME(\|x\|^n)$	The set of all decision problems solvable by a deterministic Turing machine in polynomial time.
(in general) $DTIME(f(\|x\|))$	The set of all decision problems solvable by a deterministic Turing machine in $O(f(\|x\|))$ time.
NP $NTIME(\|x\|^n)$	The set of all decision problems solvable by a non-deterministic Turing machine in polynomial time.
(in general) $NTIME(f(\|x\|))$	The set of all decision problems solvable by a non-deterministic Turing machine in $O(f(\|x\|))$ time.
$NP - hard$	The set of all decision problems which every NP problem has a polynomial time reduction to.
$NP - complete$	The set defined as the intersection: $NP \cap NP - hard$.
AP	The set of all decision problems solvable by an alternating Turing machine in polynomial time.
(in general) $DSPACE(f(\|x\|))$	The set of all decision problems solvable by a deterministic Turing machine in $O(f(n))$ space.
(in general) $NSPACE(f(\|x\|))$	The set of all decision problems solvable by a non-deterministic Turing machine in $O(f(n))$ space.
R	The set of decision problems solvable in finite time.

Table 1: A small set of different complexity classes. All of these classes are based on the Turing machine.

Cellular automaton is also studied in complexity theory. An example of how cellular automaton is studied is the way that Stephen Wolfram has divided the elementary cellular automaton rules, based on the way that the simulations run and the resulting shape, as follows[3]:

Class	Description
Class 1	Nearly all initial patterns evolve quickly into a stable, homogeneous state. Any randomness in the initial pattern disappears.
Class 2	Nearly all initial patterns evolve quickly into stable or oscillating structures. Some of the randomness in the initial pattern may filter out, but some remains. Local changes to the initial pattern tend to remain local.
Class 3	Nearly all initial patterns evolve in a pseudo-random or chaotic manner. Any stable structures that appear are quickly destroyed by the surrounding noise. Local changes to the initial pattern tend to spread indefinitely.
Class 4	Nearly all initial patterns evolve into structures that interact in complex and interesting ways, with the formation of local structures that are able to survive for long periods of time.

Table 2: Complexity classes of elementary cellular automaton, due to Wolfram.

Class 2 type stable or oscillating structures may be the eventual outcome, but the number of steps required to reach this state may be very large, even when the initial pattern is relatively simple. Local changes to the initial pattern may spread indefinitely.

The study of complexity in cellular automaton led to some remarkable results, too. Wolfram has conjectured that many –if not all- class 4 automata are capable of universal computation. Which has been later proven for rule 110 and Conway's game of life. But, still, till these days' cellular automaton is not used for algorithm implementation. The Turing machine is still the premium model for algorithmic computation.

[3] Wolfram, S. (2002). *A new kind of science vol.5.* Champaign: Wolfram Media, Inc.

4. Conclusion:

As we have seen in this research, there is no premium model of computation, or at least not yet. What we have rather is different models for different tasks. The two models that we have talked about are both universal, so there is no task one model is not capable of. But still, sometimes one model is better at a certain task concerning the running time and the space complexity and the easiness of the implementation as a whole.

The Turing machine is a model favorable by its theoretical simplicity on one side, and its practical simplicity on the other, i.e. it is relatively easy to implement. And the best aspect of the Turing machine is the universal Turing machine, which made it the most general-purpose model.

Even though cellular automaton is not general purpose, but it is very diverse. This diversity, combined with its nature as cells and rules, made it ideal for simulation of physical, biological, and even social phenomena.

What we did in this research is to do a simple comparison between the Turing machine and cellular automaton. There are lots of aspects about these two model that we didn't mention in this research. For example, the Turing machine has over 20 variations. And furthermore, there are lots and lots of other models, such as the pointer machines, lambda calculus and more. The study of these models doesn't stop here, but will be later continued, hopefully in the NCD.

5. Figure Index

Figure	Description	Source
Figure 1	A mechanical Turing machine. Even though Turing machines are not necessarily mechanical devices, it's nice to build such a replica.	
Figure 2	A 5-generation simulation of Conway's game of life using Golly 2.5 software. The initial pattern (left) yields the second pattern after applying the rules as described. The second pattern yields the third, and so on.	Using Golly 2.5 software
Figure 3	A representation of rule 110 using wolfram notation. Each block represents a neighborhood and the state of the center cell in the next generation. The string below the blocks is the representation. It is obvious that $01101110_2 = 110$, which is the way rules are enumerated.	[5]
Figure 4	(above) A representation of rule 30 using Wolfram notation. (below) A simulation of rule 30 for 15 generations. Each generation is one line. For each generation, the new generation is directly below it.	[5]
Figure 5	The simulation of rule 30 for 200 generations (left) and 1000 generations (right) made using Wolfram Mathematica 10.2 software. As we can see, in the left side of simulations there is pattern. However, this not the case in the right side, this is why rule 30 is particularly interesting, because almost all of the 256 rules don't have this chaotic behavior.	Using Wolfram Mathematica 10.2 software
Figure 6	A working replica of the Turing machine that Alan Turing built in the 1940's, called Bombe. This replica has been built using complex electric circuits.	
Figure 7	A vortex disturbed by an obstacle, taken from Wolfram's book. Wolfram later used cellular automaton to simulate this system.	[7]
Figure 8	The cellular automaton used by Wolfram to model the vortex shown in figure 7 taken from his book. The rule for this automaton, as shown in the middle, is derived from the forces that govern this system at the atomic level. The automaton scaled out so that each vector (cell) represents the average of the 25x25 neighbor vectors, as shown in the bottom.	[7]

Figure 9	Figure 9: The evolution of the automaton as mentioned above in figure 8, in generations 10000, 20000, 30000, 40000 and 70000 respectively. As we can see the simulation comes closer and closer to the shape of the vortex in figure 7.	[7]

6. References

[1] Arora, S., & Boaz, B. (2009). *Computational complexity: a modern approach.* Cambridge University Press.

[2] Hernie, F. C., & Stearns, R. E. (1966). "Two-tape simulation of multitape Turing machines". *Journal of the ACM (JACM) 13.4* .

[3] Savitch, W. J. (1970). "Relationships between nondeterministic and deterministic tape complexities.". *Journal of computer and system sciences 4.2.*

[4] Weisstein, E. "Cellular Automaton." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/CellularAutomaton.html

[5] Weisstein, E. "Elementary Cellular Automaton." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/ElementaryCellularAutomaton.html

[6] Weisstein, E. "Turing Machine." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/TuringMachine.html

[7] Wolfram, S. (2002). *A new kind of science vol.5.* Champaign: Wolfram Media, Inc.