8-1-2012

Modeling of Cellular Automata and Agent-Based Complex Systems

Sara Pallekonda
University of Nevada, Las Vegas, sarah.sush@gmail.com

MODELING OF CELLULAR AUTOMATA AND AGENT-BASED COMPLEX SYSTEMS

by

Sara S. Pallekonda

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

School of Computer Science
Howard R. Hughes College of
Engineering
The Graduate College

University of Nevada, Las Vegas
August 2012

ABSTRACT

by

Sara S Pallekonda

Dr. Woflgang Bein, Examination Committee Chair

Professor of Computer Science

University of Nevada, Las Vegas

The term ‘complex systems’ may sound terrifying whenever you come across it as it depicts an overall collective structure which indeed can live upto its name; but when you comprehend the system at its fundamental level by stripping to its simpler multiple-interacting individual parts, the insights it provides may be used to describe and understand different problems ranging from atomic particles to the economics of societies and evolution. The simple laws can be used to simulate the behaviors of disparate complex systems.

In this thesis, a brief study is done emulating few such complex systems through programming techniques like cellular automata and neural networks. The patterns of complex behavior obtained are also classified respectively along with the help of Conway’s game of life; the working of an autonomous and self organizing organism is simulated in a program written to show the complex patterns formed by a virtual ant. Then an important aspect of competition and cooperation among these agents is shown through game theory and dilemmas which throws light on the essence of survival of complex systems. A formal study is also done on the uses of artificial neural networks as associative memories and pattern recognizers.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and gratitude to my advisor Dr. Wolfgang Bein for motivating, guiding and supporting me throughout the entire duration of the thesis work. I am also grateful to my committee members, faculty, librarians, and all the other staff at University of Nevada, Las Vegas

I would also take this opportunity to show my appreciation towards the love of my parents Mr. Sunil Pallekonda, Mrs. Suhasini Pallekonda and my brother Richard. I am also indebted by the blessings showed by my grandparents.

I would like to thank all my friends especially Abhishek Biruduraju, Anuj Nayyar, and Shruthi Yarram who have motivated and supported me during my work and in my life.

TABLE OF CONTENTS

List of Tables

List of Figures

CHAPTER 1
INTRODUCTION AND OVERVIEW

Introduction

Simple things can produce complex behavior. This can be observed from the molecular level to the global level. Complex systems are things that consist of many similar and simple parts that interact. We can understand the behavior of the individual parts that make up the system easily but slightly difficult when it is seen as a whole. Thus Complex systems can be seen as collection of elements at various levels which can keep changing states. The changes are discernible not through a single rule nor reducible to a single level of explanation. The levels include rules which cannot predict the outcome from the current specifications. Systems with little interaction among the parts fall into static patterns; while on the other hand; overactive systems are embroiled in chaos. Between these two extremes is a region of criticality in which some very interesting things happen.

The complex systems can be a collection of chemical sets; cellular regulation through gene excitation and inhibition; multi-cellular animals; collective super organisms such as ant colonies, beehives, flocks of birds, school of fish, oceanic reefs; large collection of organisms such as eco-system, economies and societies.

Though all of these may look simple their complex system results in a rich and complex behavior (extremely simple rules build a very complex pattern). The science of complex systems is a rapidly evolving area, in terms of both domains and methods. The interest in this area, as well as its rather subsequent diffusion, has been rather remarkable. At the most basic level, the field of complex systems challenges the notion that by perfectly understanding the behavior of each component part of a system we will then understand the system as a whole. Our work focuses on simple examples that are accessible, yet also contains much deeper foundational insights. [1]

Complexity characteristics

The complexities arising from the huge data are characterized by the given below factors

- Self organization
- Non-linearity
- Order/chaos
- Emergent properties

Self Organization

Order can also be regarded as information, so we can classify the complexity of a system by how much information we need to describe it. The essence of self-organization is that system structure often appears without explicit pressure or involvement from outside the system. In other words, the constraints on form of interest to us are internal to the system, resulting from the interactions among the components and usually independent of the physical nature of those components. The organization can evolve in either time or space, maintain a stable form or show transient phenomena. The formation of patterns over time or space for previously independent variables operate under local rules.

The field of self-organization seeks general rules about the growth and evolution of systemic structure, the forms it might take, and finally methods that predict the future organization that will result from changes made to the underlying components. The results are expected to be applicable to all other systems exhibiting similar network characteristics. The features of such systems are autonomous, time evolution, fluctuations, global order, symmetry breaking, criticality, redundancy, self-maintenance, and adaptation.

Examples

Natural Selection is best known from the theory of evolution which described the success and extinction of species in their fight for survival. Such processes can be seen in non organic systems too. Researchers have developed computer programming techniques that solve problems based on the complex processes of biological evolution and natural selection. These techniques of "Evolutionary Computation" are called "*artificial life*" and "*genetic algorithms*".

Another pattern for Self-Organization is found for example in the complex system of the central nervous system of animals. In this example, the brain cell networks that are the ones that most successfully help the animal survive are the ones that are the most used, and thus are the ones that grow the most in size and complexity. In contrast, those brain cell networks that do not help the animal survive are less used, and thus grow less, and may even stop growing, atrophy, and disappear. The computer programming technique based on this approach to solving problems is called "*neural networks*".

A further example of a pattern of Self-Organization is where an overall task is accomplished by breaking it down into mini-tasks which are then spread among separate little parts for execution, which then also coordinate together where necessary to support the overall functioning.

Non-Linearity

The changes which occur in complex system are non linear in fashion. In non-linear change, one can see elements being changed by previous elements, but then in turn these changed elements can affect the elements that are before it in the sequence. Thus we study the possibilities on the effects that can affect anything before and after it, the results being disproportional to the original inputs. These dynamic changes are seen in nature and are quite unpredictable sometimes.

To control these non linear systems, they have already tried to make use of linear approximations effectively. The operations of predictions are done in regions where the system behaves almost linearly. This means restricting the parameters of the system to areas that do not possess the sensitivity to initial conditions or studying only simplified aspects of systems.

Order/Chaos

The uncertainty of predictability is called chaos. It becomes more and more difficult to predict what will develop based only on previous knowledge, even when that knowledge is extensive. Thus even though there is logical development from stage to stage, there is an increasing inability to predict what will actually be the next development.

A tiny change becomes increasingly difficult to predict exactly which result will actually occur. But since some probability of occurrence for many of them can be known, statistical analysis is still very important for helping describe the overall situation. A famous example is about how the flapping of butterfly wings in one part of the world can contribute to the evolution of a hurricane in another part of the world.

Emergent Properties

As the complex systems are unpredictable in nature, they can evolve giving results that are totally unpredictable even based on the initial original conditions. Such unpredictable results are called emergent properties. Emergent properties thus show how complex systems are inherently creative ones. These are logical results not just a predictable ones.

In other words we cannot predict the outcome from studying only the fine details. Examples include cellular metabolism, ant colonies, organism development, and snowflakes. [2]

Programming Techniques:

We have seen above that all these complex systems can be emulated through computer programmer techniques which can be used for analyzing, simulating, and modeling these characteristics. They are

- Cellular automata
- Artificial life
- Neural networks

CELLULAR AUTOMATA: Cellular Automata is a computing approach centered on simple programming sub-routines (called agents) which are given certain operational limitations. It is a computing machine which is a dynamical system which is discrete in both space and time, patterned after the way cells operate as parts of an organ in the body.

ARTIFICIAL LIFE: The Simulations run on a computer with computer generated entities of that type with known formulae and incorporating them into a program, running it to see what happens over time.

NEURAL NETWORKS: The programming approach is patterned after the developmental processes of the nervous systems. The result is merely more probable than the alternatives, the system just chooses that result with the highest likelihood. [2]

CHAPTER 2

CELLULAR AUTOMATA (CA)

The chess-board is the world; the pieces are the phenomena of the universe; the rules of the game are what we call the laws of Nature.

---*T. H Huxley*

Cellular Automata

Take a board, and divide it up into squares, like a chess-board or checker-board. These are the '*cells'*. Each cell has one of a finite number of distinct colors. (We don't allow continuous shading, and every cell has just one color.) Now we come to the '*automaton'* part. Sitting somewhere to one side of the board is a clock, and every time the clock ticks the colors of the cells change. Each cell looks at the colors of the nearby cells, and its own color, and then applies a definite rule, the *'transition rule'*, specified in advance, to decide its color in the next clock-tick; and all the cells change at the same time. (The rule can say "Stay the same.") Each cell is a sort of very simple computer - in the jargon, a *finite-state automaton* - and so the whole board is called a *'cellular automaton'*, or CA. To run it, you color the cells in your favorite pattern, start the clock, and stand back. [3]

To put the whole concept in a technical way, a *'Cellular Automata'* is a computing machine which is a dynamical system, discrete in both space and time. Or it can also be defined as a collection of cells arranged in a grid such that each cell changes states(which can be colors, or bits, or something abstract) as a function of time according to a defined set of rules that includes the states of neighboring cells, known as the *'transition rule'*. These nearby cells make up what is known as the *'neighborhood'* of the cell. [1]

The idea of cellular automata was first introduced by Stanislaw Ulam in the 1940s where he suggested Von Neumann to use cellular spaces to build his cell reproductive machine. Thus Neumann concentrated on the simplest mathematical framework, which would allow information to reproduce rather than on how animals reproduce. Von Neumann was able to prove that a certain CA can have a *'general constructive automaton'*, a configuration of states which can construct almost any configuration of states.[4]

In the subsequent sections, we will discuss about

- *One dimensional Cellular Automata*, which may look simple but it can exhibit different types of complex behavior
- *Wolfram's Cellular Automata classification*: It consists of 4 classes and we will discuss about their patterns.
- *Conway's game of life*, a two-dimensional automaton that is capable of emulating any Turing machine and is therefore capable of any universal computation.

Finally we conclude with a summary of natural phenomena that can be modeled by cellular automata.

2.1 One dimensional CA

One-dimensional cellular automata are the simplest realization of any automata. We can assume a linear grid of a fixed length, consisting of various cells each of which can be in any one of the k finite states at a time instant. For a cell at index i, the state is described by a variable $c_i(t)$ at the time instant t. At the next instant $t+1$, the state of the cell at index i, depends upon the neighborhood of the cell. The radius r, which determines the state of cell at index i is used to define the transition rule. It means that the state $c_i(t+1)$ is determined by the cell states $c_{i-r}(t)$ *to* $c_{i+r}(t)$, a total of $(2r+1)$ states. [5]

The simple forms of the one-dimensional cellular automata are where there are a low number of possible states in which the cells can be, normally two states i.e. $k=2$, usually represented by bits – 0 and 1. Sometimes colors are also used to represent the states, white for 0 and black for 1. The simplest one-dimensional cellular automata have the minimum possible neighborhood of 3, i.e. $r=1$, which means only the state variables $c_{i-1}(t)$, $c_i(t)$ and $c_{i+1}(t)$ determine the next time instant state $c_i(t+1)$. These automata are known as *'elementary cellular automata'*, a term coined by S. Wolfram, which have been extensively studied for their amazing properties.

For one-dimensional cellular automata, its history can be defined on a two dimensional grid. Moving left or right of the grid corresponds to moving in space where as time flows in the downward direction. The cells below the first state represent the next states.

There are 256 such automata, each of which can be indexed by a unique binary number whose decimal representation is known as the "rule" for the particular automaton. An illustration of rule 30 is shown above together with the evolution it produces after 15 steps starting from a single black cell. [6]

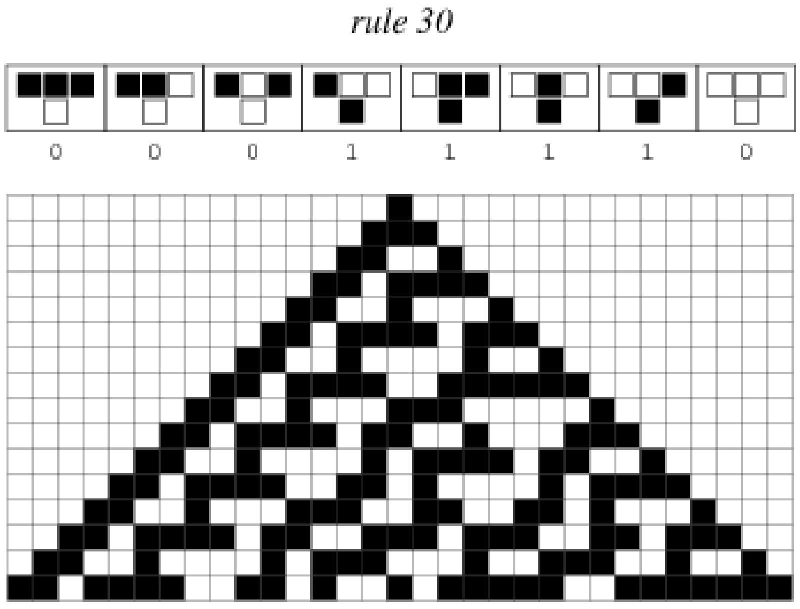


Figure 2.1.1 Rule 30, one-dimensional CA

The transition rule for the rule 30 cellular automata can be described in the following table, normally known as the *look-up table*.

$c_{i-1}(t)$	$c_i(t)$	$c_{i+1}(t)$	$c_i(t+1)$
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	1
0	1	0	1
0	0	1	1
0	0	0	0

Table 2.1.1 Look-up table for rule 30 CA

For a general case of one-dimensional cellular automata, a look-up table should have k^{2r+1} entries for k finite number of states and radius r for the neighboring cells.

$c_{i-1}(t)$	$c_i(t)$	$c_{i+1}(t)$	$c_i(t+1)$
1	1	1	0
1	1	0	1
1	0	1	1
1	0	0	1
0	1	1	1
0	1	0	1
0	0	1	1
0	0	0	0

Table 2.1.2 Look up table for another elementary CA

Another example of elementary cellular automata is discussed below. The look up table for these automata can be shown as –

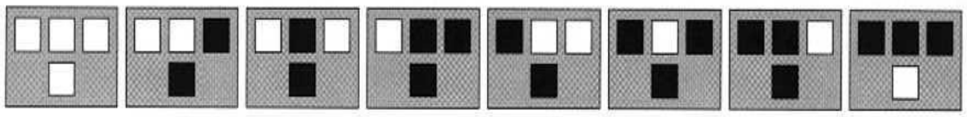


Figure 2.1.2 Graphical Representation of rules in Elementary CA

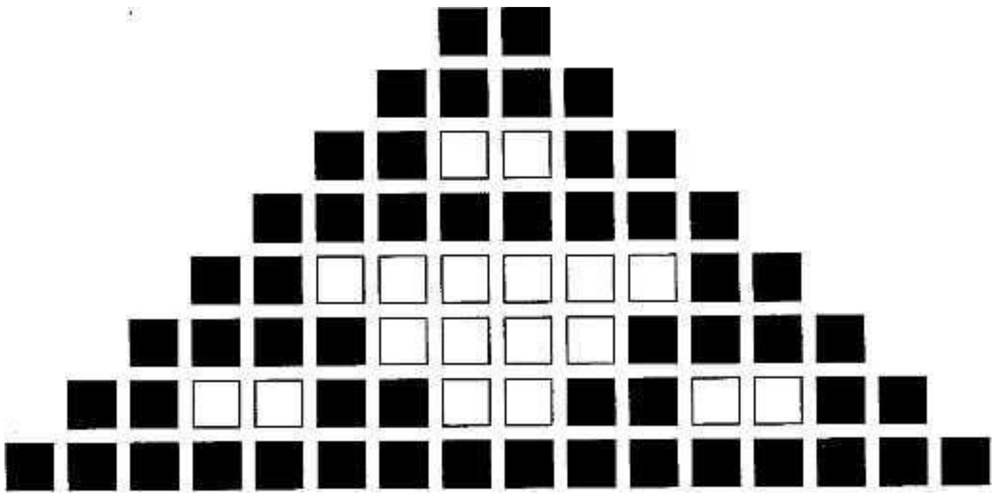


Figure 2.1.3 Resulting CA from look-up table in Elementary CA

The above table can be also stated as follows: if all of the cells in a neighborhood are on or all of the cells in a neighborhood are off then the next state is off else the next state will be on. If the sum of the states is equal to 1 or 2 then the CA yields a 1 else 0.

We can minimize the size of a rule table if we consider only the sum of the states in a neighborhood. There should be *k(2r+1)* entries in a rule table to analyze each possible total sum.

If we examine a CA with more than two states where the rule's table length is *k(2r+1)* it can be represented by *k(2r+1)* digits. Using the above rule table we will simulate the steps of a CA evolving over a period of time as shown below:

Let the initial state be "11" which doubles its size to "1111" after one-time step. Let us consider each digit as a cell; we see that the center two cells have neighboring states of *on*, so the next state must be *off* according to the table. Likewise we continue subsequently for each step.

The numbering system for elementary CA

There are possible configurations for a cell when we consider its two immediate neighbors to be involved in the transition rule. This leads to a possibility of

elementary cellular automata. A scheme called *Wolfram Code* was proposed by Stephen Wolfram, to assign a number from 0 to 255 to define these schemes. Each possible current configuration is written on order, 111, 110, ... , 001, 000, and the resulting state for each of these configurations is written in the same order and interpreted as the binary representation of an integer. This number is taken to be the rule number of the automaton. For an example the 30 in binary is 00011110_2. So the rule 30 is defined by the transition rule shown in table below.

Current pattern	111	110	101	100	011	010	001	000
New state of center cell	0	0	0	1	1	1	1	0

Table 2.1.3 Look up table for rule 30 CA

2.2 Wolfram's CA classification

The types of patterns that are generated from one-dimensional CAs for a ring of n cells with k states will be of the order of n^k which will be large but finite. So wolfram researched on one-dimensional CA's and he classified CA's into four different types. [7]

- *Class I* - CAs in this class always evolve homogeneously with every cell being in the same state.

- *Class II* - CAs in this class form periodic structures that cycle endlessly through a fixed number of states.
- *Class III* – CAs in this class form “aperiodic” random like patterns.
- *Class IV* – CAs in this class form complex patterns with localized structures that move through space in time. These patterns must eventually become homogeneous like Class I or periodic like Class II.

Class I: They can be simple programs which die after a few execution steps and their behavior is not very interesting. There is no initial configuration of sites which produces patterns of any length using these rules; it will always evolve to a homogeneous state. They are similar to dynamical systems, which have a fixed point solution; i.e. no matter what input you start with, they always end up in a fixed state in a few steps.

A few examples of elementary cellular automata, which are Class I are rule 250 and rule 254.

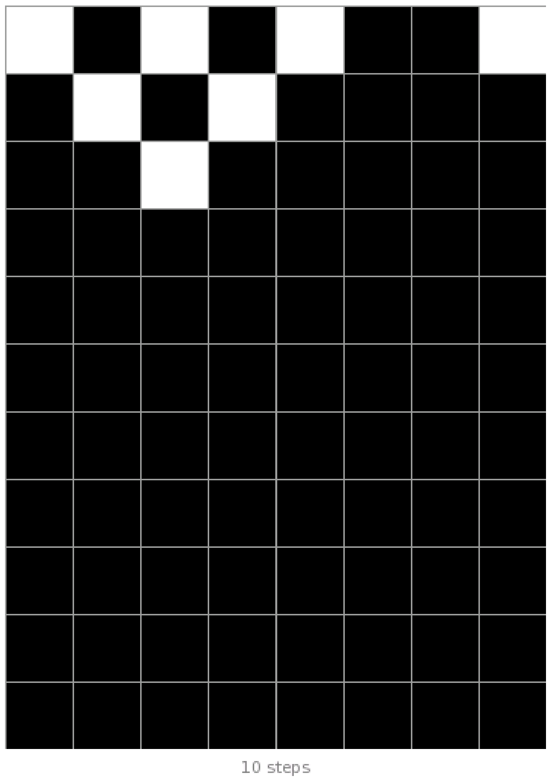


Figure 2.2.1 Evolution of rule 250 CA (Class I)

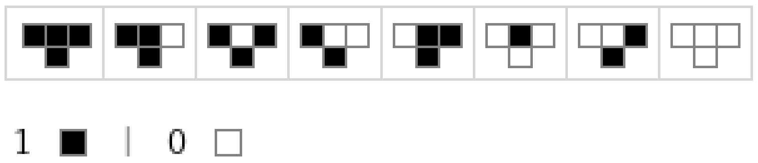


Figure 2.2.2 Graphical representation of rules in rule 250 CA

Class II: They are repetitive automata that can be infinite loop programs. In terms of dynamical systems, they are the ones that fall into the category, which have limit cycles. There are two types of patterns in this class, "simple periodicity" whose behavior is captured in a finite space and "unbounded periodicity" whose behavior wraps around continuously in an infinite state never returning to any previous state but can be limited to a periodic behavior in a finite sized space.

A few examples of elementary cellular automata, which are Class II are rule 4 and rule 108.

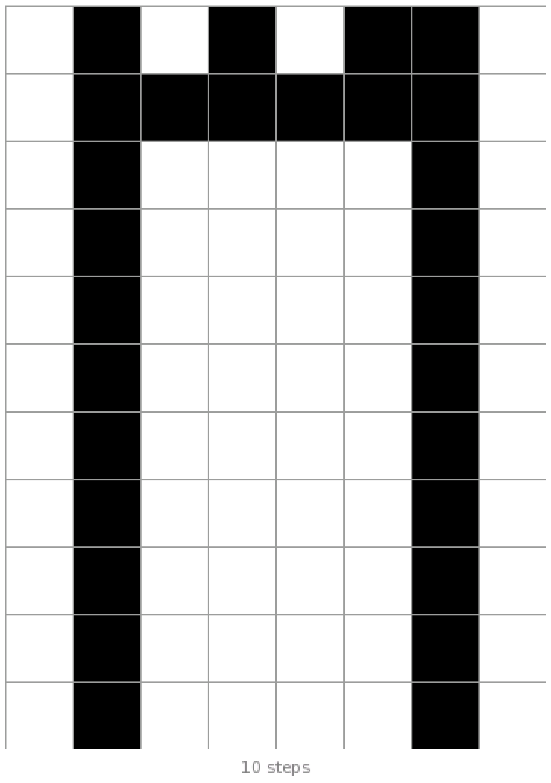


Figure 2.2.3 Evolution of rule 108 CA (Class II)

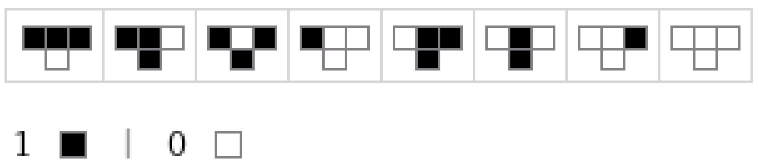


Figure 2.2.4 Graphical representation of rules in rule 108 CA

Class III: They can be chaotic dynamical systems, where in a finite space they never repeat and repeat only in long cycles. These classes of CAs also vary according to the initial conditions where an altered initial state can result in a totally different pattern from the previous unaltered one. We can get a similar behavior of Class II when we embed limit cycles of shorter duration but these can be highly unstable and resort back to chaotic behavior with slight disturbances.

As with all well-defined chaotic systems, a given state is completely reversible; the previous state (and all the ones before) can be predicted by examining the current state. This class of automata is non-deterministic in that to find the value of a site after an infinite amount of time, an infinite number of initial conditions must be considered. The Class III CA is very similar to the programs that are used as pseudo-random number generators. With just a slight change in the initial configuration of cells, a very dramatic change can be observed in the grid pattern as time progresses. This causes such automata to be highly unstable, as they are very sensitive to slight variations.

A few examples of elementary cellular automata, which are Class III are rule 30 and rule 90.

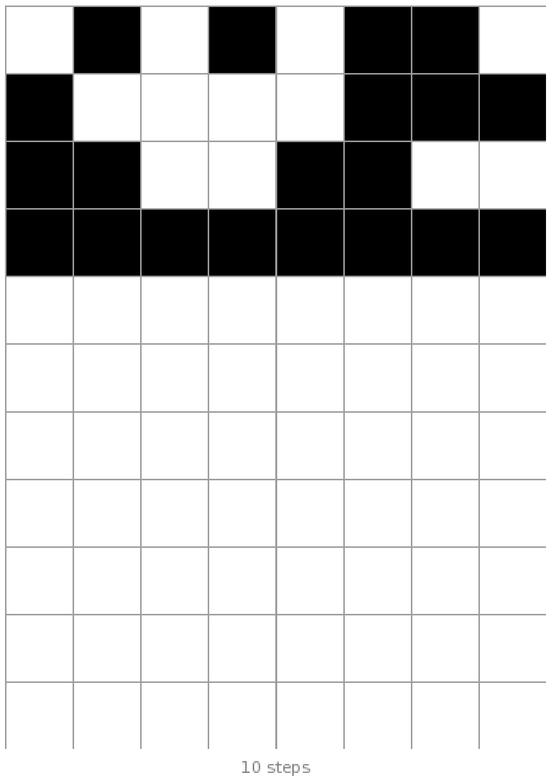


Figure 2.2.5 Evolution of rule 90 CA (Class III)

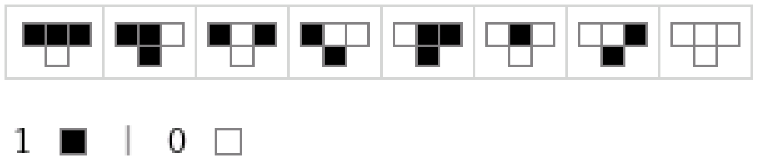


Figure 2.2.6 Graphical representation of rules in rule 90 CA

Class IV: These are difficult to describe as its behavior is a bit of regular, periodic or random. Its behavior moves in between producing some chaotic and some regular patterns, and are thus localized, stable, but non-periodic configurations. Given the right initial conditions and a finite number of time steps, any computable problem can be solved by it.

These configurations can be seen as *encoding* packets of information, *preserving* them through time, and *moving* them from one spatial location to another: information can propagate in time and space without undergoing important decay. The amount of unpredictability in the behavior of $Class_4$ rules also hints at computationally interesting features: by the Halting Problem, it is a key feature of universal computation that one cannot in principle predict whether a given computation will halt given a certain input. These insights led Wolfram to conjecture that Class IV CA were (the only ones) capable of universal computation. Intuitively, if we interpret the initial configuration of a Class IV CA as its input data, a universal Class IV CA can evaluate any effectively computable function and emulate a universal Turing machine. [8]

A few examples of elementary cellular automata, which are Class IV are rule 54 and rule 110.

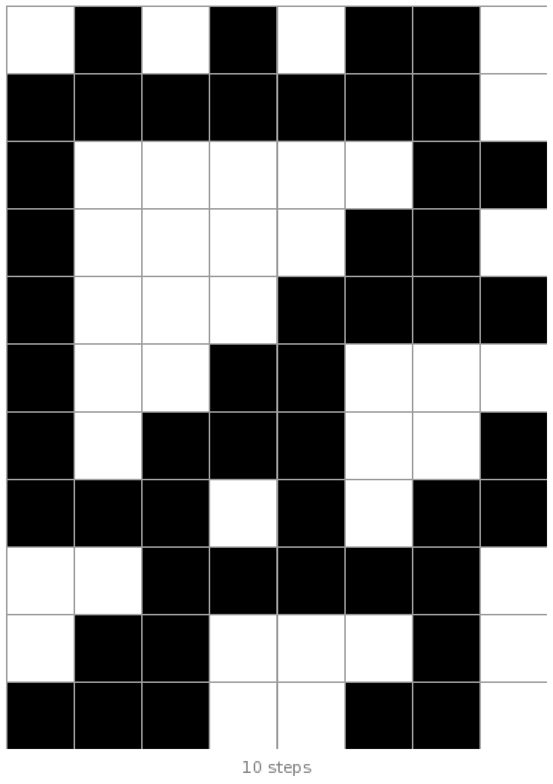


Figure 2.2.7 Evolution of rule 110 CA (Class IV)

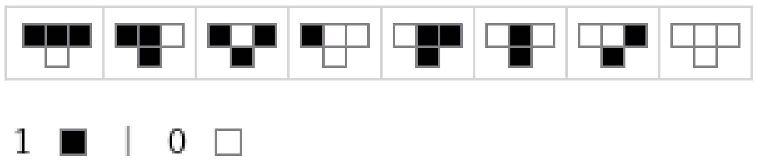


Figure 2.2.8 Graphical representation of rules in rule 110 CA

2.3 Conway's Game of Life

Let us take an infinite check board. As we all know that it works on very simple rules where we replace the current checker to form a new interesting pattern. We can create a Turing- equivalent computing machine by placing the checkers cleverly and by interpreting the patterns appropriately. These interesting patterns helped Conway and his students to believe that Life could support universal computation. Conway's work has become popular by Martin Gardener in his *Scientific American* column. [9]

John Conway had redefined the description of a cellular automaton using only two states, on and off, which could determine the next state using simple rules. It is an infinite two dimensional cellular automaton. He named it as "The Game of Life" because the two states were analogous to "live" and "dead" and the rules used were "realistic".

Let us assume that each square of the board is a cell and each cell has eight neighbors. The next state of a cell is a function of the states of nearest neighboring cells and the immediate neighbors of a cell which form the boundary are the eight cells in each direction which we name them as N, S, E, W, NE, NW, SE, and SW. The rules for time evolution are:

- Loneliness- if a live cell has less than two neighbors, then it dies
- Overcrowding- if a live cell has more than three neighbors, then it dies
- Reproduction- if an empty cell has three live neighbors, then it comes to life
- Stasis- if it has exactly two live neighbors, then the cell stays as it is.

Given an initial random configuration, we can see the evolution of the system performing universal computations as it is in Langton's critical area.

Examples of patterns:

1. *Static objects* - these do not change over time. As they are fixed persistently, they form a basic type of memory. Let us take a simple 2x2 filled square grid. Here it is clear that each live cell has exactly three live neighbors and hence it cannot go to the next generation. And each cell outside this cluster has one or two live neighbors and hence cannot reproduce to form a new cell. Such patterns are known as *static objects* in Conway's Game of Life. [10]

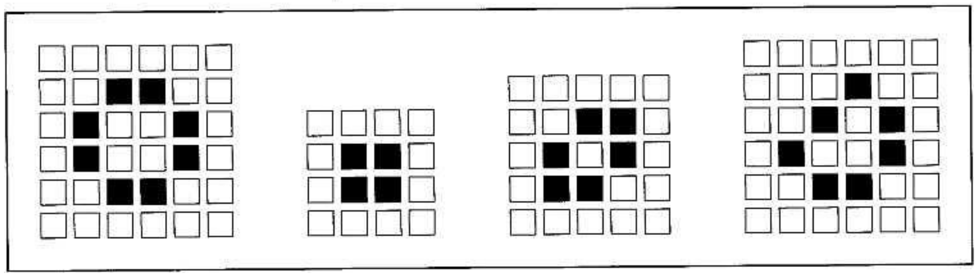


Figure 2.3.1 Examples of static objects in Conway's are game of life

2. *Periodic objects* - these are used to synchronize events parallel in time and coordinate iterative operations. Periodic structures consist of three live cells in a row alternatively between three vertical and horizontal cells. The next generation cells are deceased by loneliness or overcrowding and new cells are formed which are again crushed. There is a mutual placement which is enough to reproduce a similar configuration in the next generation. .These repeating patterns are similar to oscillators. [10]

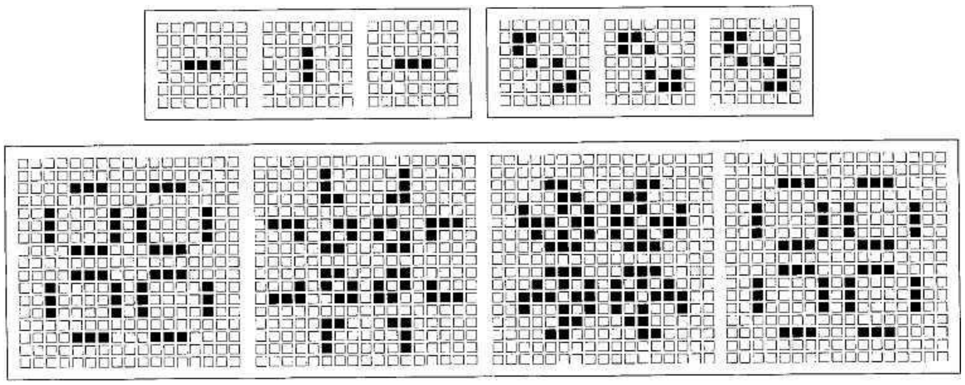
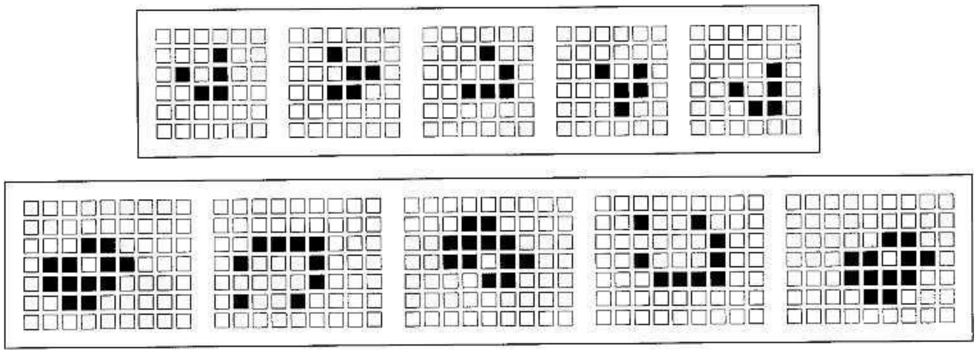


Figure 2.3.2 Examples of simple periodic objects in Conway's Game of Life

3. *Moving objects* – these have the ability to move information, which Conway realized as the substitute to von Neumann's wires. Some frequently occurring examples are gliders, fish and spaceships. They can move in any of the eight possible directions. Gliders are one of the first to be found and these are formed by only five live cells. Hence the consistency of their prevalence. Self propulsion is cable by only Gliders and three small space ships. These can be combined to form other objects which collide with other type of objects in some ways, like in a *breeder*. [1]

Figure 2.3.3 Examples of moving objects in Conway's game of life



There is a possibility of self replicating machines when the system assembles a copy of itself and sets in motion assembling pieces to form larger and complex objects which can be self reproducing. As they are capable of self replication, they can also be made to perform universal computations from simple logical primitives of NOT, OR, and AND for arguments A and B.

2.4 Real world CA examples

We come across complex systems in our daily life. Some of the examples are”

- Statistical mechanical systems – before supercomputers were invented, scientists were satisfied with purely statistical description of stochastic systems but now the scientists use a powerful computer to experiment by programming the laws of physics into a computer model to simulate. For example a lattice gas automaton is a model of how gas or plasma interacts in a local space where each molecule cares about its nearest neighbors and nothing else.
- Autocatalytic chemical sets- it is a collection of chemicals that is self catalyzing and therefore capable of highly non linear dynamics. Each molecule in a chemical reactant can interact with other molecules which are located near the first molecule. Local interactions combined with parallel evolution of the simultaneously occurring chemical reactions, yield remarkably accurate pictures of how these systems self organize.

- Gene regulation- a single gene can activate another gene which in turn activates another gene and so on. Here genes are not inhibited by space in terms of affecting other genes unlike autocatalytic chemical sets.
- Multi-cellular organisms – a multi-celled organism is formed by pooling several single celled organisms which increases its efficiency to survive.
- Colonies and super-organisms – species in the insect world such as ants, termites, bees and wasps form colonies. They combine concepts of parallelism, specialization, local interactions with limited autonomy.
- Flocks and herds – birds move in a flock for their safety. They face two sub-problems, which they solve by maintaining certain distance from other birds since they don't want to be too close. They try to be in the middle of flock to avoid being attacked by the predators. Similar behavior can be seen in fish and herds too.
- Ecosystems – there are diverse species in an ecological system which has to be stable to display a rich variety of population dynamics. Any disturbances to it drastically affect the ecosystem.
- Economies and society – systems have to be competitive and cooperative to be controlled economically. Anarchical systems fail in competing with systems that cooperate on limited scale.

CHAPTER 3

AUTONOMOUS AGENTS AND SELF-ORGANISATION

Autonomy means a self governing state, community or groups who have a unique capability of surviving. Autonomous agents are independent agents and they do not take commands from other agents though it interacts with its environment and other agents too. They can solve tasks in a spread out fashion where multiple agents coordinate, cooperate and compete. Self organization represents a fundamental reallocation of energy and action within a system in order to achieve a larger goal. It results in unexpected complex behavior from simple rules through which stable patterns emerge. Self-organization: this type of behavior is seen in many cases like chemical soups, gene regulation systems, super organisms, animal collectives, economical systems etc where multiple agents can perform tasks which look global.

Let us examine few examples where such phenomena is observed

3.1 Termites

Termites are small insects which that roam around randomly. If we observe their behavior we can observe the following rules:

- Wander around aimlessly, via a random walk, until the termite bumps into a wood chip.
- If the termite is carrying a wood chip, it drops the chips and continues to wander.
- If the termite is not carrying a wood chip, it picks up the one that it bumped into and continues to wander.

These rules are very simple to follow. Let us take a small number of termites on a square grid on where the wood chips are randomly placed. We can see that initially while simulating the termites, they move around randomly moving chips back and forth and then form small clusters of wood chips whenever they bump into a wood chip and gradually after few more steps they form defined large clusters. Final plot of well defined collection can be seen after thousands of time steps.[11]

The other rules that can be used on termites to prevent wastage of time and to form a collection of single large pile of chips are:

- The termites make a random left or right and then take a step instead of just taking a random step in any direction.
- After dropping or picking up a wood chip, the termites would always make a 180 degree turn and hence they can avoid picking up the same piece they dropped.

Many other changes can be added to improve the efficiency of the termites.

3.2 Virtual Ants

Let us take a look upon a single virtual ant on a grid where all the points are either white or black which can generate different type of complex behavior and the ant wraps from edge to edge.

Rule set of ant for each time step:

- The ant takes a step forward.
- If the ant is now standing on a white point, then it paints the point black and turns 90 degrees to the right.
- Otherwise, if the ant is standing on a black point, then it paints it white and turns 90 degrees to the left.

Using these rules we get eight steps that an ant starts initially from a blank grid. We know that the CAs are time reversible and so is the case for Langton's rules. We can determine an ant's future and also its past steps. For eg if the ant is standing on a black cell, then we know that at the previous time step the cell was white and that the ant just made a 90 degree turn to the right. Though time reversibility seems to be simple globally too, it is not true since it does not necessarily imply global simplicity as we can see from the given eight steps that after few steps the ant interacts with the grid locations where it has been in the past which shows that the ant acts recursive. The entire future could be changed if a cell's color from the past is flipped at some point.[12]

Suppose we simulate the ant for another 10,000 steps or so we can observe a chaotic looking image with no structure but after that the ant reaches a fixed pattern. James Propp discovered that after another 250 steps or more the ants starts to build a phenomenon called a highway. The ant will build a highway for ever if there is an infinite cell space but practically it will eventually intersect a place in the ant’s grid space where it has been in the past and then again we can observe the chaotic behavior for few steps, but after another few steps the ant will again build the highway. Thus if the number of states that the ant and the grid can be are finite, then the ant will fall into some sort of cyclic pattern.

Suppose we have two ants in a grid space then at some point because of the irreversible nature, both will move in a single cell space and collide with one another in such a way that they start to undo the others work.

We get a situation in the grid where the configuration is same as some previous state but the ants positions are swapped and so is the direction they face (opposite in this case). If the ants are roaming around for a while without intersecting, then they are cleaning up the cell space by a stage of un-building highways and removing random patterns. After collision since they are swapped in positions and directions, they go on to build another highway system. This type of simulation is oscillating repeats in a cyclic fashion.[13]

When more ants are allowed to travelled in the cell space even more interesting things happen and a common occurrence would be building of partial un-built highways. Apart from varying the number of ants we could also simulate the paths by having an initial configuration of the grid which is not empty, and we could have *n* states instead of just two. The *n*-bit rule string can thus determine the behavior of an ant. If we have an ant with a rule string of all 1s or all 0s, then the ant travels around in a little square. Also if any of the ant has a rule string consisting of at least one 1 or 0, it cannot be confined in finite spaced grid.

3.3 Flocks, Herds, and Schools

We study how agents by themselves produce uninteresting results, but produce a variety of behaviors when interacting with similar agents. These agents can be a collection of animals such as flocks, herds etc which move about in space in a composed pattern.

A model was created which simulated the motion of flock of birds called *boids*. Each of these follows a set of rules to optimize various goals. The goals that they try to achieve are intuitive. Some of the rules are:

- *Avoidance* – movement of boids in such a way so as to reduce the chances of collision
- *Copy* – move in the direction of a flock by estimating the other boids velocities and directions.
- *Center* – moving to the center of the flock so as to avoid exterior exposure
- *View* – move laterally away from any boid that blocks the view.

Avoidance is one of the most important rules and cannot be ignored by any boid. Copy and center rules are inactive when the avoidance rule is being followed by a boid as it is difficult for the boid to simultaneously attempt to avoid, and to copy or center on, any other boids that are too close.

Copy rule strengthens the boids to stick together over the long term for several reasons such as being in a flock keeps them safe and to stay with their mate etc., collisions will be reduced if the birds follow this rule as they will maintain same velocities and head in the same direction. We allow the boids to have a look at other boids with a fixed angle because in reality they cannot really know the velocities of each and every other boid in the flock. Another constraint in this rule is that their vision is limited to a finite distance.

Center rule is a very greedy rule where every boid thinks only about their safety at the cost of other boids and wants to be in the center of the flock to avoid attacks from their predators. Even in center rule the vision of boids is fixed to a certain angle and distance but we allow them to use a different viewing radius for the purpose of averaging neighbors' positions.

View rule is not a mandatory rule, it just seemed like a good idea where we partially blind the boids with an unrealistic viewing angle to form a "V" by the flock. The view rule works by moving the boid in a direction perpendicular to the vector that joins the first boid and the boid that is interfering with its sight. Since there are two such paths, the boids select the direction that is closer to the path where the flock is originally heading. This rule is faced with an unrealistic effect where in the boid never seems to slow down. The visual interference defines a narrow region by an angle and distance.[14]

Now we combine the above goals into a single action, say a single direction which is obtained by taking the weighted average of the four directions. If *v* terms are velocity vectors, *w* terms are weighting factors, and μ is the momentum (to make the boids move along the previous path without changing direction) term, we get

$$V_{new} = \mu V_{old} + (1-\mu)(W_{avoid}V_{avoid} + W_{copy}V_{copy} + W_{center}V_{center} + W_{view}V_{view})$$

With the new composite velocity vector and the old position, we get the new position as

$\square_{new} = \square_{old} + \tau V_{new}$ where □ terms are positional vectors and τ denotes a time increment.

The ordering of the four weights is

$W_{avoid} > W_{view} > W_{center} > W_{copy}$ because collision avoidance is more important than copying.

Examples of boids in motion:

1. If the view rule is disabled, we see that a disorganized boids combine to form a single flock

2. If we give larger weights to center and avoidance rules while turning off the view rule, we get to see that boids circle each other in a cyclic manner. The smaller boids start moving in an eccentric manner for sometime as they catch a glimpse of the bigger boid and eventually join the bigger swarm.

CHAPTER 4

COMPETITION AND COOPERATION

Even though there is a lot of competition universally among organisms, the need to cooperate with each other arises for a better trade of survival. For example, symbiosis which is seen even in humans where bacteria co-exist to help in digestion and it also benefits by getting its food. Also many slime molds exhibit cooperation to combine and form a large multi-cellular organism when there is a scarcity of food. This is seen even in higher class of animals. Cheating in such scenarios is not viable because their success is closely related with cooperation. [15]

Axelrod, a scientist examined the necessary requirements for how cooperation evolves in a competitive environment, how stable it is, how robust and a profitable strategy it can lead to be. This was studied using Game Theory and we look at how few of the games described below emulate the tendencies as shown by organisms. The prisoner's dilemma of game theory is a very good example of such strategy.

The game is defined by a set of strategies and a payoff function which determines the payoff a player receives when playing its strategy against the strategy of another player. Every individual follows a particular strategy and the payoff of this strategy will determine the reproduction rate of that individual in relation to other strategies. This ultimately regulates the fraction of players with a particular strategy, since a player never changes his strategy and passes his strategy to his offspring in a reproduction event.

4.1 Non zero sum games and dilemmas

We study about games where a possibility arises for cooperation of players, and where there are win-win or lose-lose results; such games are called non-zero sum games which are realistic. Where as in zero-sum games, there is no universally accepted solution. That is, there is no single optimal strategy that is preferable to all others, nor is there a predictable outcome. Non-zero-sum games are non-strictly competitive, as opposed to the completely competitive zero-sum games, because such games generally have both competitive and cooperative elements. Players engaged in a non-zero sum conflict have some complementary interests and some interests that are completely opposed. [16]

4.2 Prisoner’s Dilemma

This game is about the interaction between the two players based on an understanding of motives and strategies. Let us take two suspects who have been arrested for a crime.

Cooperation is a term where both suspects are silent to protect each other from conviction as the police don’t have enough evidence to put them in jail. With the evidence that they have in their hand, the suspects can get a sentence of say two years.

Defection is something where each suspect is asked separately to confess their crime for freedom or reduced sentence. if either one of the suspect confesses then the other suspect who doesn't will get ten years in jail and the defector will be rewarded with freedom. If both confess, both get five years in jail.

There are many outcomes for these two cases. Cooperation between both players is the most benefitting choice. Because getting 10 years is not a favorable outcome if one of them doesn’t cooperate and if both the players doesn’t cooperate then each will get a 5 year sentence which is not the best outcome either but for a situation like this the best strategy would be to for both to confess. In fact there is a more chance of confession by one of them. When the players decide to cooperate as the player defecting can gain freedom.[17]

By observing all the above outcomes we can say that when two partners in crime who are guilty have been arrested. The best option for any player is defecting no matter what the opponent does.

Table 4.2.1 Pay-off Matrix for prisoner’s dilemma

	Keep quiet	Confess
Keep quiet	3,3	1,4
Confess	4,1	2,2

Where 4 represents a player’s most favored outcome and 1 is the least favored outcome.

A strategy is said to be strictly dominant if that strategy’s pay off value for a player is greater than the other.

Applications of Prisoner’s Dilemma

1. Two countries considering whether to go to war: if we replace the above two words i.e. keep quiet with defend , and confess with attack, we can see that countries are better off defending and each country’s worst outcome is to be defensive while the other attacks.

2. International trade: a country deciding whether to levy tax against other country’s goods or not.

3. Advertising of products in a duo-polized market competition.

Strict dominance is powerful methodology in game theory and to focus on each player’s dominant strategy we have to consider one payoff at a time. When an individual player in a game evaluates separately each of the strategy combinations he may face, and, for each combination, choose from his own strategies the one that gives the best payoff. If the same strategy is chosen for each of the different combinations of strategies the player might face, that strategy is called a "dominant strategy" for that player in that game.

If each player has a dominant strategy, and if each player plays the dominant strategy, then that combination of (dominant) strategies and the corresponding payoffs are said to constitute the dominant strategy equilibrium for that game.

In the Prisoners' Dilemma game, to confess is a dominant strategy, and when both prisoners confess, that is dominant strategy equilibrium.[18]

Stag hunt: Two hunters are in a hunting range without any communication and they have to choose whether to hunt hares or the stag. If both of them are after the hares then each captures half of the hares. If one of them hunts for the stag then he is left empty handed as the other one hunts all the hares. And if both of them are after the stag, their share of stag is greater than the value of all hares.

Table 4.2.2 The payoff matrix for Stag hunt is given as follows:

	Stag	Hare
Stag	3,3	0,2
Hare	2,0	1,1

This game lacks a dominant strategy and hence we look for Nash equilibrium. It is a set of strategies, for each player such that it has no effect on the other player's doings. So in this example <stag, stag> is Nash equilibrium which is also referred to as pure strategy Nash equilibrium (PSNE) because both hunters are playing deterministic strategies. If iterated elimination of strictly dominated strategies reduces the game to a single outcome, that outcome is Nash equilibrium and it is the only Nash equilibrium of that game.

The stag hunt unlike the prisoner's dilemma can analyze interdependent decisions i.e. each hunter's individual optimal strategy is a function of other hunter's choice. It also shows how Nash equilibrium need not be efficient as <hare, hare> can also be seen as a sustainable outcome because neither hunter has the opportunity to change depending on the other's doing.

Now we look at mixed strategy Nash equilibrium. In the prisoner's dilemma and stag hunt, the players found it beneficial to cooperate with each other whereas in games which do not have a pure strategy, players want to see the other perform poorly. Many examples fit this type of payoffs; a soccer penalty kick (striker can kick left or right, he wants the goalkeeper to dive in the opposite direction), a cricketer can throw a fast or slow ball (batsman can guess either), in football the offense can choose a running play or passing play the defense can choose to protect against the pass or run.

Such a strategy can be seen in a daily life example where mom injects fairness among siblings by allowing one child to slice the cake and the other to choose either of the two pieces, and thus the child tries to slice the cake exactly into two equal pieces.

These are called zero sum games. The primary goal of game theory is to provide a player with the best possible strategy for a particular game. Now we look at one such game as described below

Matching Pennies

Let us consider a game where two players are given a penny to be placed on a table with either side facing up. If both pennies are of the same side then the first player (player A) receives them or else the second player (player B) gets them

	Heads	Tails
Heads	1,-1	-1,1
Tails	-1,1	1,-1

Table 4.2.3 Payoff matrix for Matching Pennies

The rules for the game can be described in the payoff matrix above. The best strategy is to randomly pick heads or tails with equal probability. This is mixed strategy because player should mix things up when there is absence of information. To randomize things for mixed strategies, players can decide to flip the coins before placing them and thus the probability induced reduces the chances of pure strategies.

Now the generalized expected score for a player is

E(A) = HH × PA(heads) × PB(heads) + HT × PA(heads) × (1 – PB(heads)) + TH × (1 – PA(heads)) × PB(heads) + TT × (1- PA(heads)) × (1- PB(heads))

Where <HH>, <HT>, <TH>, and <TT> correspond to the four possible payoffs for a player; PA, PB are probabilities of playing a head by respective players. This equation will have at least one equilibrium which means that a best strategy can be known by looking for flat spots on the surface plot. This equilibrium is known as mixed strategy Nash equilibrium.

4.3 Iterated Prisoner's Dilemma

The classical IPD is used to study the evaluation of co-operation. Numerous strategies can be generated by genetic approaches.

It is identical to the previous version except that the players play for many rounds, and also players has an idea about their opponent's previous moves. Thus decisions can be based on whether they have been cooperative or not. Iteration plays an important role.[19]

To have a dilemma, the following inequation has to be achieved:

Temptation >Reward >Punishment >Sucker's payoff

Robert Axelrod assigned numerical values to each of the four possible playoffs so as to keep a score of the winner; 5 points for temptation payoff (DC=5), 3 points for mutual cooperation (CC=3), 1 point as a punishment for mutual defection (DD=1), and 0 points as sucker's payoff (CD=0). Some of the simple unrealistic strategies:

- all_c: Always cooperates. [c]*
- all_d: Always defects. [d]*
- tit_for_tat: The tit_for_tat strategy was introduced by Anatole Rapoport. It begins to cooperate, and then play what its opponent played in the last move.
- Spiteful: It cooperates until the opponent has defected, after that move it always defects.

- soft_majo: Plays opponent's majority move, if equal then cooperates. First move is considered to be equality.
- per_ddc: Plays periodically : [d,d,c]*
- per_ccd: Plays periodically : [c,c,d]*
- mistrust: Defects, then plays opponent's move.
- per_cd: Plays periodically [c,d].
- Pavlov: The win-stay/lose-shift strategy was introduced by Martin Nowak and Karl Sigmund. It cooperates if and only if both players opted for the same choice in the previous move.
- tf2t: Cooperates except if opponent has defected two consecutive times.
- hard_tft: Cooperates except if opponent has defected at least one time in the two previous move.
- slow_tft: Plays [c,c], then if opponent plays two consecutive time the same move plays its move.
- hard_majo: Plays opponent's majority move, if equal then defects. First move is considered to be equality.
- Random: Cooperates with probability ½.

To study the behavior of these strategies, two kinds of computation can be done.[20]

1. Round-Robin Tournament: Here each strategy meets all other strategies. The sum of all scores in each confrontation is its final score. Range in the tournament is used at the end to get the measurement of the strategy's strength. This way Axelrod has isolated the tit-for-tat strategy.

2. Simulated Ecological Evolution: The population of bad strategies is decreased whereas good strategies obtain new elements from the fixed initial population using the Round-Robin Tournament made. Once the population is stabilized or does not change anymore, the simulation stops. We can see that strategies which try to defect initially are not stable and good ones take over. These increasing good ones then stay for longest possible time in the population.

The weakness of the above strategies is that they lack memory i.e. they do what they do regardless of pervious occurrences of both players. Another strategy that was introduced in the tournament was TFT (tit for tat), which punishes any defections later while being nice at first i.e. cooperates in the first round of any game of IPD then it will do exactly what the opponent did in the previous round. Its same as ALL-C when played against it but against ALL-D, it gets beaten first but will reciprocate with defection for the rest of the game.

To be good a strategy must:

- Be nice, i.e. not be the first to defect
- Be reactive
- Forgive

- Not be too clever, i.e. to be simple in order to be understood by its opponent

4.4 IPD Applications for Complex Systems

- The Iterated Prisoner's Dilemma (IPD) is widely used to study the evolution of cooperation between self-interested agents. Existing work asks how genes that code for cooperation arise and spread through a single-species population. We study the competition between different species of agents as a macro-evolutionary phenomenon. The more selfish species tend to get extinct while competing. The outcome of this competition depends on many factors: chance, population size, the species and their initial proportions in the population. By manipulating some of these, we aim to understand better how cooperative behavior can evolve in populations of self-interested individuals, and what factors affect that evolution.[21]
- We have seen above that TFT strategy performed well in the aforementioned tournaments in a 2-person IPD, but what is the best strategy in an N-person IPD? With the advancements in technology, every player's history information of choice could be seen by all other players. A new tournament is presented to study the strategy selection for NSIPD. [22]

The model for this method is that it has an infinite population of players and there are a finite number of distinct strategies in this population. Some of which were used for experimentation were –

TIT FOR TAT (TFT); ALLD (always Defect disregarding the opponent' Defect’s history); ALLC (always cooperate disregarding the Cooperate opponent's history); RAN (Randomly cooperate or defect regarding the opponent's history); Never Forgive (NF); F_1 (forgive once, Cooperate if opponent defects only once in latest three moves, otherwise defect); F_2 (forgive twice, Cooperate if opponent defects only two times in latest three moves, otherwise defect).

This has shown that TFT does not have overwhelming advantage over other six strategies and the best strategy lies on round number and strategy distribution. The second result is that ALLD performs worst synthetically. There is no theoretical best strategy and how to do well lies on many factors

CHAPTER 5

NEURAL NETWORKS

Natural and Analog Computation

The artificial neural networks have been evolving to achieve human like performance in the fields of image and speech recognition which are one of the greatest potential of neural networks. This is a vast subject where there are many examples, equations, simulations, theorems, implementations that are being studied.

While this is an impractical technique for very complicated complex problems, it still demonstrates how nature and the laws of physics can work together in order to solve interesting problems. There are many other types of phenomena that can be seen as solving optimization problems, if viewed in the right way.

Associative memory and combinatorial optimization problem are two main applications of neural networks. Our brain is the best example for associative memory.

The soap bubble can be used as a comparison for distributed dynamical system that can compute interesting things. Here the whole soap film wants to minimize its surface area and not just one bubble. Each molecule in a soap solution interacts only with a relatively small number of neighboring molecules. Hence, a global property surface area is minimized by only local interactions. Similarly, global properties such as the collection of neural activations that compose a distributed memory or the solution to an optimization problem may emerge from only local interactions.[1]

5.1 Artificial Neural Networks

A typical neuron has a cell body, many dendrites and an axon that ends with a bundle of terminal fibers. A neuron operates by receiving signals from other neurons through connections, called synapses. In human brain there are around one hundred billion neurons which are connected to another thousand different neurons through the dendrites or the cell body itself. The dendrites are receptors for signals generated by other neurons. Information is propagated through these connections by sending a pulse through the axon.

The combination of these signals, in excess of a certain threshold or activation level, will result in the neuron firing, which is sending a signal onto other neurons connected to it. If a neuron fires an electrical impulse is generated. The impulse starts at the base, called the hillock, of a long cellular extension, called the axon, and proceeds down the axon to its ends.

Artificial neural network is nothing but a collection of artificial neurons which are interconnected and share the properties of biological neural networks. It is a network of simple processing neurons with limited connectivity which exhibit interesting complex behavior.[22]

During the early stages neural networks remained very uncertain for nearly 20 years. The McCulloch and Pitts model of a neuron has been very important in computer science as it was one of the first. In fact, you can buy MCP neuron at most electronic stores, but they are called "threshold logic units." A group of MCP neurons that are connected together is called an **artificial neural network**. Threshold logic unit is a very simplified model of a neuron.

In McCulloch-Pitts neural model, the neurons state or activation $a_i(t)$, at time t, is a function of a weighted sum of all of the incoming signals and in order to excite or a fire a neuron at any time, a certain number of synapses must be excited within a period of latent addition. The position of the neuron and its previous activity does not affect this number.(if it is greater than the predetermined threshold then it will fire with activation of 1, else the activation of neuron is 0.) an output signal is either 0 i.e., discrete value or a real value number between 0 and 1 The activation function has a sigmoid shape.

$$a_i(t+1) = \Theta(\sum_{j=1}^{n} w_{ij} * a_j(t) - b_i)$$

Where -

ai(t) is activation value of neuron I at time t

wij is strength of synapse connecting neuron j to neuron i

bi is threshold that neurons i’s net input must exceed in order to fire

Θ(x) is unit step function: 1 if x ≥ 0, 0 if x < 0

Sgn(x) is sign function: 1 if $x \geq 0$, -1 if $x < 0$

Hi is net input, ∑jwijaj(t), into neuron i

This model has been updated but still it has its limitations as it cannot be taken as real. It just reflects one or more neurophysiologic observations. At a given time step, the activation values can be updated in two ways. One is synchronous and the other asynchronous updating.

Using a directed graph we can represent artificial neural network which consists of many neurons that are interconnected.

Here is an example of a directed graph with a n ordered tuples (V,E), where V is the set of vertices that represents neurons and E is the set of edges that represent the synaptic strengths i.e., the weights attached.[23]

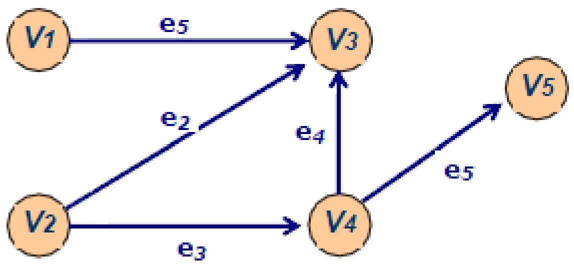


Figure 5.1 Example of a directed graph

Vertices V = { v_1,v_2,v_3,v_4,v_5 }

Edges   E = { e_1,e_2,e_3,e_4,e_5 }

5.2 Associative Memory and Hebbian Learning

One of the most integral parts of our cognitive function is memory. Computers are being used extensively for modeling. Simulation is a very important tool to organize and synthesize a coherent understanding from massive amounts of experimental data. Hebb's theory is one of the most earliest and realistic theories on associative memory. Associative memory's function is to recognize previously learned input vectors, from the know vectors even if some noise has been added to it.

Hebbian learning is a learning algorithm derived from biological neurons that can be used to train associative networks. In Hebbian learning, units that fire together wire together. Self organizing is one of the main features of Hebbian learning. [24]

Hebb's postulate states that, " when an axon of cell A is near enough to excite cell B or repeatedly, or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased". [25]

Though Hebb's principal is purely theoretical, the mathematical terms for this principal has not been formulate by him. One of the basic hebbian learning rules is formulated as,

$$\Delta w_{ij} = \varepsilon a_i a_j$$

Where, $\Delta\ w_{ij}$ denotes the change in weight from unit i to unit j,

a_i and a_j denote the activation levels of units i and j respectively,

ε denotes the learning rate.

Changes in the strength of connection weights in neural networks are an implementation of Hebbian learning. Weight changes as a function of units' activity levels in Hebbian learning.

From the above equation we can say that a_i a_j are directly proportional to the weights i.e., as the value of a_i and a_j increases so will the weights. At a given time t, the value of weight can be calculated as,

$$w_{ij}\ (t)= w_{ij}\ (t-1) + \Delta w_{ij}$$

The problem with our Hebbian rule is that though the weights can increase, they cannot decrease. Over successive steps is one of the reasons for this problem because the rule forces the weights within the network to become infinitely large. The other reason is that the weights cannot decrease accordingly. Hence by normalizing the weight updates with the following equation these problems can be solved.

$$\Delta\, w_{ij}\ (t)= \varepsilon a_j\ (a_i - w_{ij})$$

Speech and Image Recognition

These models are used in speech and image recognition processes to achieve human like performances. It consists of many nonlinear computing elements arranged in parallel and patterns emulating a biological neural net. These computational nodes are connected via variable weights that improve performance in adapting during use. These nets which act as highly parallel building blocks can be used to construct more complex systems. They also provide better robustness and fault-tolerance than Von Neumann sequential computers

Classifiers perform three tasks:

1. Identify which class represents the best input pattern (as generally they are noisy inputs)

2. Used as associative memory where the input pattern is used to determine which output has to be produced. Sometimes only a part of the pattern is available and for further retrieval there are additional stages which design Hopfield nets as content-addressable memories.

3. Perform vector quantization, used to compress the bits in speech and image recognition for faster transfer of analog data.

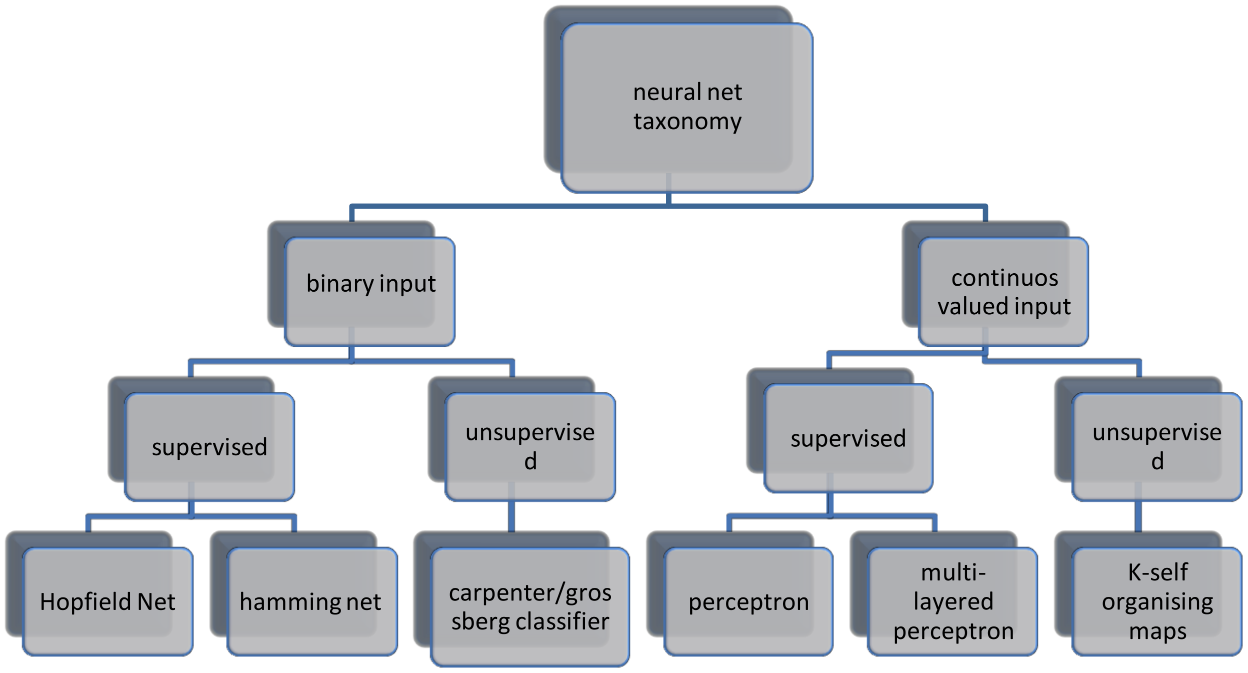


Figure 5.2 Neural net taxonomy

Hopfield Net

This can be used an associative memory to solve optimization problems. It takes N binary inputs; the output of each node is fed back to all other nodes via weights. The initial weights are set using the exemplary patterns and then when a noisy corrupt pattern is fed, the net iterates in times steps by a given formula to match the pattern. The net output after convergence is used directly as the restored memory. The limitations of this net are:

- The number of stored patterns is limited and sometimes many a stored patterns can give rise to dubious matches.
- An example pattern can be unstable if it shares many bits in common with another example pattern.

Hamming Net

As seen above the Hopfield net is used where inputs are selected randomly or reversing the bits of exemplary patterns in a probabilistic manner, but when binary fixed-length signals are sent through memory-less binary symmetric channels, the hamming bit distance is used to calculate the optimum minimum error classifier. Such a model which implements this algorithm in a neural net component is called hamming net. (The hamming distance is the number of bits in the input which do not match the example bits)

Carpenter/Grosberg Classifier

This net implements a clustering algorithm of leader selection. The leader algorithm selects the first input as the exemplar for the first cluster. The next input is compared to this first cluster. It is clustered with the first if the distance to the first is less than a threshold value or it becomes exemplary for a new cluster. This process is repeated so on. The structure is similar to hamming net and the matching scores are computed using feed-forward connections. This differs from hamming net in that feedback connections are provided from output nodes to the input nodes.

PERCEPTRON

It has the ability to recognize simple patterns. The single node computes a weighted sum of the inputs (belonging to classes), subtracts a threshold and outputs the decision as a non-linearity to determine the class. It forms two decision regions separated by a hyper plane whose equation depends on the connection weights and the thresholds. The structure can be used to implement Gaussian maximum likelihood classifier, a robust technique.

MULTI-LAYER PERCEPTRON

These are feed-forward nets with more layers of nodes between inputs and outputs. A single layer perception forms half plane decision regions whereas a two-layer perception can form any possibly unbounded convex regions. Multiple layers can form complex regions as obtained using nearest-neighbor classifiers. These can be used to create continuous likelihood functions.

KOHONEN’S SELF ORGANIZING FEATURE MAPS

These maps are similar to the placement of neurons in an orderly fashion in our brain. A two dimensional array of output nodes is used to form the feature map where each input is connected to every output node via a variable connection weight. The weights will specify clusters such that the point density of the vectors tends to approximate the probability density function of the input vectors. This is vital in complex systems reducing the lengths of inter-layer connections. This performs relatively well in noise because the number of classes is fixed, weights adapt slowly, and adaptation stops after training.

The neural nets potentiality lies in its high speed processing through massively parallel implementations. They can be used to

- Analyze and learn about self-organizations
- Analyze and learn about self-organizations
- developing design principles to solve sensitivity problems for large analog systems
- Building complete systems for image and speech recognition
- Determining which algorithm can be implemented using neuron like components.

CHAPTER 6

CONCLUSION AND RESULTS

In this thesis, we thoroughly analyzed and studied various complex systems. Examples were studied where various simple rules when applied over a large time led to really complex and sometimes unpredictable phenomena, or even settle in a final state which does not change with time thereafter.

The simplest ones among these are –

- Elementary Cellular Automata
  - Wolfram’s classification of elementary CA
- Conway’s Game of Life

Of these, we implemented the codes for all the four Wolfram classes of Cellular automata with a fixed length 8-bit random sequence, and ran the algorithm for a fixed number of steps to study the different ways in which these different classes of CA develop.

Also, a similar scenario has been implemented for the Conway’s Game of Life on a fixed size grid (rather than an infinite grid) starting with some alive cells and letting the simulation run for a fixed time to see the evolution with time.

We then moved to some more complex systems involving autonomous agents and self organization. The ones we studied here were –

- Termites
- Langton’s Virtual Ants (single and multiple)
- Flocks and Herds

The experiments in NISPD differ from Axelrod’s IPD. We have seen that TFT need not be the best over other six strategies and the best strategy lies on round number and strategy distribution; and also shown is that ALLD performs worst. Also defecting is not the preferred strategy even if people might not play game in fixed, repeated long-run interactions with certain people. There is no theoretical best strategy and performance is based on many factors. A harmonious society with cooperation and competition can be built up if these factors are constrained by technology. The evolution of strategy in NSIPD is under investigation.

The report also described a wide variety of neural network modules for associative memory, category learning, and pattern recognition. The general description of these neural nets is given but the field has been emerging rapidly and more complex systems are under development. These have higher speeds of processing through massive parallelism implementing VLSI. Future research in solving dynamic and sensitive problems for large analog systems can be implemented using neuron like components i.e. *real time neural net systems.*

By studying all of these, we analyzed various simple phenomena leading to complex behaviors over time. Also we saw that the learning algorithms can be used to solve a variety of problems in real life.

Future Work

In the future, we can work on more smart algorithms, or extending the algorithms to have graphical outputs for better understanding. Also we can choose some more real life problems, and try to implement the concepts learnt and discussed here to try and find solutions for them.

APPENDIX

A. One Dimensional Elementary Cellular Automata

The code below, written in Microsoft Visual Studio implements a generate case of elementary CA. An example of each of the four classes of CA is shown:

The code generates one-dimensional elementary cellular for an 8-bit array over 10 time steps. The initial input is taken as randomly generated bit array of length 8. The different CA, that are implemented are –

- Rule30 - A general CA
- Rule250 - A Class I CA
- Rule108 - A Class II CA
- Rule90 - A Class III CA
- Rule54 - A Class IV CA

Based on the Wolfram Code of the elementary CA, the corresponding Kernaugh Map is used to find the underlying logic; which is then used for the implementation of the cellular automata.

```
// elementaryCA.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"

const unsigned int nSize = 8;
const unsigned int nTime = 10;

unsigned int nArray[nTime][nSize] = {0};

void applyRule30();              // A general CA
void applyRule250();             // Class I CA
void applyRule108();             // Class II CA
void applyRule90();              // Class III CA
void applyRule54();              // Class IV CA

void printArray();               // Printing CA

int _tmain(int argc, _TCHAR* argv[])
{
    for (int nIndex = 0; nIndex < nSize; nIndex++)
{
                nArray[0][nIndex] = rand()%2;
        }

        applyRule30();
        printf("\n A General CA \n");
        printArray();

        applyRule250();
        printf("\n A Class I CA \n");
        printArray();

        applyRule108();
        printf("\n A Class II CA \n");
        printArray();

        applyRule90();
        printf("\n A Class III CA \n");
        printArray();

        applyRule54();
        printf("\n A Class IV CA \n");
        printArray();
```

```
          return 0;
}

void applyRule30()
{
          for(int nTimeIndex = 1; nTimeIndex < nTime; nTimeIndex++)
          {
                    for(int nIndex = 0; nIndex < nSize; nIndex++)
                    {
                              int nLeft, nCurrent, nRight;

                              nCurrent = nArray[nTimeIndex-1][nIndex];
                              if (nIndex == 0)
                              {
                                        nLeft = 0;
                                        nRight = nArray[nTimeIndex-1][nIndex+1];
                              }
                              else if (nIndex == nSize-1)
                              {
                                        nRight = 0;
                                        nLeft = nArray[nTimeIndex-1][nIndex-1];
                              }
                              else
                              {
                                        nLeft = nArray[nTimeIndex-1][nIndex-1];
                                        nRight = nArray[nTimeIndex-1][nIndex+1];
                              }

                              int nNext = (nCurrent + nRight + nCurrent*nRight +
nLeft*nCurrent*nRight)%2;
                              nArray[nTimeIndex][nIndex] = nNext;
                    }
          }
}

void applyRule250() // Class 1 exapmple Also Rule 254
{
          for(int nTimeIndex = 1; nTimeIndex < nTime; nTimeIndex++)
          {
                    for(int nIndex = 0; nIndex < nSize; nIndex++)
                    {
                              int nLeft, nCurrent, nRight;

                              nCurrent = nArray[nTimeIndex-1][nIndex];
                              if (nIndex == 0)
```

```
                        {
                                nLeft = 0;
                                nRight = nArray[nTimeIndex-1][nIndex+1];
                        }
                        else if (nIndex == nSize-1)
                        {
                                nRight = 0;
                                nLeft = nArray[nTimeIndex-1][nIndex-1];
                        }
                        else
                        {
                                nLeft = nArray[nTimeIndex-1][nIndex-1];
                                nRight = nArray[nTimeIndex-1][nIndex+1];
                        }

                        int nNext = nLeft + nRight - nLeft*nRight;
                        nArray[nTimeIndex][nIndex] = nNext;
                }
        }
}
void applyRule108() // Class 2 exapmple Also Rule 4
{
        for(int nTimeIndex = 1; nTimeIndex < nTime; nTimeIndex++)
        {
                for(int nIndex = 0; nIndex < nSize; nIndex++)
                {
                        int nLeft, nCurrent, nRight;

                        nCurrent = nArray[nTimeIndex-1][nIndex];
                        if (nIndex == 0)
                        {
                                nLeft = 0;
                                nRight = nArray[nTimeIndex-1][nIndex+1];
                        }
                        else if (nIndex == nSize-1)
                        {
                                nRight = 0;
                                nLeft = nArray[nTimeIndex-1][nIndex-1];
                        }
                        else
                        {
                                nLeft = nArray[nTimeIndex-1][nIndex-1];
                                nRight = nArray[nTimeIndex-1][nIndex+1];
                        }

                        int nNext = (nCurrent + nLeft*nRight)%2;
                        nArray[nTimeIndex][nIndex] = nNext;
                }
        }
}

void applyRule90() // Class 3 CA, Also see Rule 30
{
        for(int nTimeIndex = 1; nTimeIndex < nTime; nTimeIndex++)
        {
                for(int nIndex = 0; nIndex < nSize; nIndex++)
                {
                        int nLeft, nCurrent, nRight;

                        nCurrent = nArray[nTimeIndex-1][nIndex];
                        if (nIndex == 0)
                        {
                                nLeft = 0;
                                nRight = nArray[nTimeIndex-1][nIndex+1];
                        }
                        else if (nIndex == nSize-1)
                        {
                                nRight = 0;
                                nLeft = nArray[nTimeIndex-1][nIndex-1];
                        }
                        else
                        {
                                nLeft = nArray[nTimeIndex-1][nIndex-1];
                                nRight = nArray[nTimeIndex-1][nIndex+1];
                        }

                        int nNext = (nLeft + nRight)%2;
                        nArray[nTimeIndex][nIndex] = nNext;
                }
        }
}

void applyRule54()      // A Class 4 CA, also see Rule 110
{
        for(int nTimeIndex = 1; nTimeIndex < nTime; nTimeIndex++)
        {
                for(int nIndex = 0; nIndex < nSize; nIndex++)
                {
                        int nLeft, nCurrent, nRight;

                        nCurrent = nArray[nTimeIndex-1][nIndex];
                        if (nIndex == 0)
                        {
                                nLeft = 0;
                                nRight = nArray[nTimeIndex-1][nIndex+1];
                        }
                        else if (nIndex == nSize-1)
                        {
                                nRight = 0;
                                nLeft = nArray[nTimeIndex-1][nIndex-1];
                        }
                        else
                        {
                                nLeft = nArray[nTimeIndex-1][nIndex-1];
                                nRight = nArray[nTimeIndex-1][nIndex+1];
                        }

                        int nNext = (nLeft + nCurrent + nRight + nLeft*nRight)%2;
                        nArray[nTimeIndex][nIndex] = nNext;
                }
        }
}

void printArray()
{
        for(int nRowIndex = 0 ; nRowIndex < nTime; nRowIndex++)
        {
                for(int nColIndex = 0; nColIndex < nSize; nColIndex++)
                {
                        printf(" %d ", nArray[nRowIndex][nColIndex]);
                }
                printf(" \n ");
        }
}
```

B. Langton's Ant

The following codes written in Microsoft Visual Studio are an implementation of Langton's ant on a fixed size grid. We assume a square grid, and place an ant (or two) pointing in specific direction(s) on the grid, and run the evolution for a fixed time (twice the size of grid in this case). The resulting state of the grid at each time instant is printed showing 1's at the cells, which are in ON state; and 0's at the cells in the OFF state.

Code for single Langton ant

```
// visualAnt.cpp : main project file.

#include "stdafx.h"
#include "stdio.h"

// GRID SIZE
const unsigned int nSize = 20;

// CONSTANTS FOR DIRECTIONS, COLORS AND TURNS
enum Directions
{
	North, East, South, West
};

enum Colors
{
	White, Black
};

enum Turn
{
	turnRight, turnLeft
};

// STRUCTURE THAT DEFINES THE SINGLE BLOCK OF THE GRID
struct block
{
	int nXpos;
```

```
        int nYpos;
        int nColor;
}Grid[nSize][nSize];  // SQUARE GRID CONTAINING nSize*nSize BLOCKS

// FUNCTION DECLARATIONS
void InitializeGrid();
void SetColor(int *pXpos, int *pYpos);
void Move(int *pXpos, int *pYpos, int *pDir);

int main()
{
        // SETTING THE GRID AND ANT POSITION AND ORIENTATION INITIALLY
        InitializeGrid();

        int nCurrentX = nSize/2;
        int nCurrentY = nSize/2;
        int nDirection = North;
        int nTurn = turnRight;

        // POINTERS TO UPDATE VALUES

        int *pCurrentX  =  &nCurrentX;
        int *pCurrentY  =  &nCurrentY;
        int *pDirection = &nDirection;

        // FILE TO SHOW THE CHANGE IN GRID OVER ITERATIONS
        FILE * pFile;
        pFile = fopen("visualAnt.txt", "w");

        // MOVING THE ANT 2*nSize TIMES
        for(int nIndex = 0; nIndex < 2*nSize; nIndex ++)
        {
                // PRINTING THE CURRENT GRID STATE
                for(int nYIndex = 0; nYIndex < nSize; nYIndex++)
                {
                        for(int nXIndex = 0; nXIndex < nSize; nXIndex++)
                        {
                                printf(" %d ", Grid[nXIndex][nYIndex].nColor);
                                fprintf(pFile,    " %d ", Grid[nXIndex][nYIndex].nColor);
                        }
                        printf("\n");
                        fprintf(pFile, "\n");
                }
                printf("******************************************");
                printf("\n\n");
                fprintf(pFile,"******************************************");
```

```
			fprintf(pFile, "\n\n");

			// CHECKING AND CHANGING THE COLOR OF CURRENT BLOCK
			SetColor(&nCurrentX, &nCurrentY);

			// SETTING THE TURNING DIRECTION AND ORIENTATION
			if(Grid[nCurrentX][nCurrentY].nColor == Black)
			{
				nTurn = turnRight;
				if(nDirection == West)
					nDirection = North;
				else
					nDirection++;
			}
			else
			{
				nTurn = turnLeft;
				if(nDirection == North)
					nDirection = West;
				else
					nDirection--;
			}

			// MOVING THE ANT
			Move(&nCurrentX, &nCurrentY, &nDirection);
		}

		fclose(pFile);

		getchar();
   return 0;
}

// FUNCTION DEFINITIONS
void InitializeGrid()
{
		for(int nYIndex = 0; nYIndex < nSize; nYIndex++)
		{
			for (int nXIndex = 0; nXIndex < nSize; nXIndex++)
			{
				Grid[nXIndex][nYIndex].nXpos = nXIndex;
				Grid[nXIndex][nYIndex].nYpos = nYIndex;
				Grid[nXIndex][nYIndex].nColor = White;
			}
		}
}
```

```
void SetColor(int *pXpos, int *pYpos)
{
        if(Grid[*pXpos][*pYpos].nColor == White)
                Grid[*pXpos][*pYpos].nColor = Black;
        else
                Grid[*pXpos][*pYpos].nColor = White;
}

void Move(int *pXpos, int *pYpos, int *pDir)
{
        switch (*pDir)
        {
        case North:
                (*pYpos)--;
                break;
        case East:
                (*pXpos)++;
                break;
        case South:
                (*pYpos)++;
                break;
        case West:
                (*pXpos)--;
                break;
        default:
                break;
        }
}
```

Code for two Langton ants

```
// visualAnt.cpp : main project file.

#include "stdafx.h"
#include "stdio.h"

// GRID SIZE
const unsigned int nSize = 20;

// CONSTANTS FOR DIRECTIONS, COLORS AND TURNS
enum Directions
{
	North, East, South, West
};

enum Colors
{
	White, Black
};

enum Turn
{
	turnRight, turnLeft
};

// STRUCTURE THAT DEFINES THE SINGLE BLOCK OF THE GRID
struct block
{
	int   nXpos;
	int   nYpos;
	int nColor;
}Grid[nSize][nSize];  // SQUARE GRID CONTAINING nSize*nSize BLOCKS

// FUNCTION DECLARATIONS
void InitializeGrid();
void SetColor(int *pXpos, int *pYpos);
void Move(int *pXpos, int *pYpos, int *pDir);

int main()
{
	// SETTING THE GRID AND ANT POSITION AND ORIENTATION INITIALLY
	InitializeGrid();

	// For Ant 1
	int nCurrentX = 2*nSize/3;
int nCurrentY = 2*nSize/3;
int nDirection = North;
int nTurn = turnRight;

// For Ant 2
int nCurrentX2 = nSize/3;
int nCurrentY2 = nSize/3;
int nDirection2 = North;
int nTurn2 = turnRight;

// POINTERS TO UPDATE VALUES

// For Ant 1
int *pCurrentX = &nCurrentX;
int *pCurrentY = &nCurrentY;
int *pDirection = &nDirection;

// For Ant 2
int *pCurrentX2  =  &nCurrentX2;
int  *pCurrentY2  =  &nCurrentY2;
int *pDirection2 = &nDirection2;

// FILE TO SHOW THE CHANGE IN GRID OVER ITERATIONS
FILE * pFile;
pFile = fopen("visualAnt.txt", "w");

// MOVING THE ANT 2*nSize TIMES
for(int nIndex = 0; nIndex < 2*nSize; nIndex ++)
{
        // PRINTING THE CURRENT GRID STATE
        for(int nYIndex = 0; nYIndex < nSize; nYIndex++)
        {
                for(int nXIndex = 0; nXIndex < nSize; nXIndex++)
                {
                        printf(" %d ", Grid[nXIndex][nYIndex].nColor);
                        fprintf(pFile,    " %d ", Grid[nXIndex][nYIndex].nColor);
                }
                printf("\n");
                fprintf(pFile, "\n");
        }
        printf("****************************************");
        printf("\n\n");
        fprintf(pFile,"****************************************");
        fprintf(pFile, "\n\n");

        // CHECKING AND CHANGING THE COLOR OF CURRENT BLOCK
        SetColor(&nCurrentX, &nCurrentY);
        SetColor(&nCurrentX2, &nCurrentY2);

        // SETTING THE TURNING DIRECTION AND ORIENTATION
        if(Grid[nCurrentX][nCurrentY].nColor == Black)
        {
                nTurn = turnRight;
                if(nDirection == West)
                        nDirection = North;
                else
                        nDirection++;
        }
        else
        {
                nTurn = turnLeft;
                if(nDirection == North)
                        nDirection = West;
                else
                        nDirection--;
        }

        if(Grid[nCurrentX2][nCurrentY2].nColor == Black)
        {
                nTurn = turnRight;
                if(nDirection2 == West)
                        nDirection2 = North;
                else
                        nDirection2++;
        }
        else
        {
                nTurn = turnLeft;
                if(nDirection2 == North)
                        nDirection2 = West;
                else
                        nDirection2--;
        }

        // MOVING THE ANTS
        Move(&nCurrentX, &nCurrentY, &nDirection);
        Move(&nCurrentX2, &nCurrentY2, &nDirection2);
}

fclose(pFile);

getchar();
```

```
    return 0;
}

// FUNCTION DEFINITIONS
void InitializeGrid()
{
        for(int nYIndex = 0; nYIndex < nSize; nYIndex++)
        {
                for (int nXIndex = 0; nXIndex < nSize; nXIndex++)
                {
                        Grid[nXIndex][nYIndex].nXpos = nXIndex;
                        Grid[nXIndex][nYIndex].nYpos = nYIndex;
                        Grid[nXIndex][nYIndex].nColor = White;
                }
        }
}

void SetColor(int *pXpos, int *pYpos)
{
        if(Grid[*pXpos][*pYpos].nColor == White)
                Grid[*pXpos][*pYpos].nColor = Black;
        else
                Grid[*pXpos][*pYpos].nColor = White;
}

void Move(int *pXpos, int *pYpos, int *pDir)
{
        switch (*pDir)
        {
        case North:
                (*pYpos)--;
                break;
        case East:
                (*pXpos)++;
                break;
        case South:
                (*pYpos)++;
                break;
        case West:
                (*pXpos)--;
                break;
        default:
                break;
        }
}
```

C. Conway’s Game of Life

The code below, written in Microsoft Visual Studio is an implementation of Conway’s Game of Life on a fixed size grid. At the start of evolution, some of the cells are put in the ON state, and the algorithm is allowed to run for a fixed amount of time. The current implementation computes the output on a 10*10 grid, for 10 time steps. The resulting state of the grid at each time instant is printed showing 1’s at the cells, which are in ON state; and nothing is displayed at the cells in the OFF state.

```
// conway.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "stdio.h"

// Parameters to define size of grid and number of times the loop runs
const unsigned int nWidth = 10;
const unsigned int nHeight = 10;
const unsigned int nLoop = 10;

// Function Declarations
void clearArray(int nArray[][nWidth]);
void initializeArray(int nArray[][nWidth]);
void printArray(int nArray[][nWidth]);
void calculateNext(int nArray1[][nWidth], int nArray2[][nWidth]);
void swapArray(int (*nArray1)[nWidth], int (*nArray2)[nWidth]);

int _tmain(int argc, _TCHAR* argv[])
{
        // Two arrays - one to store current state, other to store the computed next state
        int nCurrent[nWidth][nHeight], nNext[nWidth][nHeight];

        clearArray(nCurrent);
        initializeArray(nCurrent);

        // Updating the grid nLoop times, and printing the values
        for (int nIndex = 0; nIndex < nLoop; nIndex++)
```

```
	{
		printf("Time Cycle = %d\n", nIndex+1);
		printArray(nCurrent);
		calculateNext(nCurrent, nNext);
		swapArray(nCurrent, nNext);
	}

	return 0;
}

// Setting all values to OFF state
void clearArray(int nArray[][nWidth])
{
	for (int nIndex1 = 0; nIndex1 < nHeight; nIndex1++)
	{
		for(int nIndex2 = 0; nIndex2 < nWidth; nIndex2++)
			nArray[nIndex1][nIndex2] =  0;
	}
}

// Setting some ON state inputs in grid
void initializeArray(int nArray[][nWidth])
{
	nArray[4][5]  = 1;
	nArray[4][6]  = 1;
	nArray[5][4]  = 1;
	nArray[3][4] = 1;
}

//	Printing the grid
void printArray(int nArray[][nWidth])
{
	for (int nIndex1 = 0; nIndex1 < nHeight; nIndex1++)
	{
		for(int nIndex2 = 0; nIndex2 < nWidth; nIndex2++)
		{
			if(nArray[nIndex1][nIndex2] == 1)
				printf(" %d ", nArray[nIndex1][nIndex2]);
			else
				printf("   ");
		}
		printf("\n");
	}
}

// Calculating the state of grid at next time
```

```
void calculateNext(int nArray1[][nWidth], int nArray2[][nWidth])
{
	unsigned int nNeighbours;
	for (int nIndex1 = 0; nIndex1 < nHeight; nIndex1++)
	{
		for(int nIndex2 = 0; nIndex2 < nWidth; nIndex2++)
		{
			// Counting Neighbors
			nNeighbours = 0;
			if(nArray1[nIndex1-1][nIndex2] == 1)
				nNeighbours++;
			if(nArray1[nIndex1+1][nIndex2] == 1)
				nNeighbours++;
			if(nArray1[nIndex1][nIndex2-1] == 1)
				nNeighbours++;
			if(nArray1[nIndex1][nIndex2+1] == 1)
				nNeighbours++;
			if(nArray1[nIndex1-1][nIndex2-1] == 1)
				nNeighbours++;
			if(nArray1[nIndex1-1][nIndex2+1] == 1)
				nNeighbours++;
			if(nArray1[nIndex1+1][nIndex2-1] == 1)
				nNeighbours++;
			if(nArray1[nIndex1+1][nIndex2+1] == 1)
				nNeighbours++;

			// Setting new values in temporary array
			if(nArray1[nIndex1][nIndex2] == 1 && nNeighbours < 2)
				nArray2[nIndex1][nIndex2] = 0;
			else if(nArray1[nIndex1][nIndex2] == 1 && nNeighbours <= 3)
				nArray2[nIndex1][nIndex2] = 1;
			else if(nArray1[nIndex1][nIndex2] == 1 && nNeighbours > 3)
				nArray2[nIndex1][nIndex2] = 0;
			else if(nArray1[nIndex1][nIndex2] == 0 && nNeighbours == 3)
				nArray2[nIndex1][nIndex2] = 1;
		}
	}
}

// Updating the grid with newly calculated values
void swapArray(int (*nArray1)[nWidth], int (*nArray2)[nWidth])
{
	for (int nIndex1 = 0; nIndex1 < nHeight; nIndex1++)
	{
		for(int nIndex2 = 0; nIndex2 < nWidth; nIndex2++)
		{
```

```
                        int nTemp = nArray1[nIndex1][nIndex2];
                        nArray1[nIndex1][nIndex2] = nArray2[nIndex1][nIndex2];
                        nArray2[nIndex1][nIndex2] = nTemp;
                }
        }
}
```

REFERENCES

[1]. Gary William Flake. “*The Computational Beauty of Nature*”

[2]. David Kirshbaum. “*Introduction to Complex Systems*”.

[3]. Cellular automata. *http://www.cscs.umich.edu/~crshalizi/notebooks/cellular-automata.html*

[4]. Boccara, and Nina. “*Modeling Complex Systems (2003*)”

[5]. Burks Arthur W, and John von Neumann. *“Theory of Self-Reproducing Automata* (1966)”

[6]. Cellular Automaton. *“http://reference.wolfram.com/mathematica/ref/CellularAutomaton.html”*

[7] Farmer J. Doyne, Toffoli Tommasso, and Wolfram Stephen. “*Cellular Automata, Los Alamos,* (1983)”

[]. Harold V. McIntosh. “*One dimensional Cellular Automata*”

[8]. Stephen Wolfram. “*Universality and Complexity in Cellular Automata, Physica 10D, 1-35 (1984)”*

[9]. Robert Bosch and Michael Trick. "*Constraint Programming and Hybrid Formulations for Life*"

[10]. Andrew Adamatzky. "*Game of Life Cellular Automata*".

[11]. Resnick, M. "Turtles, termitesand traffic jams: Explorations in massively parallel microworld" (1994)

[12]. Langton, C. "Studying artificial life with cellular automata" Physica D,(1986)

[13]. Olivier Beuret and Marco Tomassini. "*Behavior of Multiple Generalized Langton's Ants*".

[14]. Reynolds, C. W. "Flocks, herds, and schools: A distributed behavioral model"

[15]. Axelrod, and Hamilton. "*Evolution of cooperation*".

[16]. William Spaniel. "*Game theory".*

[17]PD. "http://science.howstuffworks.com/game-theory1.htm"

[18]. IPD. "http://math2033.uark.edu/wiki/index.php/Prisoner's_Dilemma"

[19]. IPD. "http://www.lifl.fr/IPD/ipd.frame.html"

[20]. Robert Axelrod. "*Evolution of strategies in iterated prisoner's dilemma*".

[21]. Philip Hingston. "*Iterated Prisoner's Dilemma for species*".

[22]. Sihai Zhang, Min Xu, and Xufa Wang. "*Computer tournaments in n-person stochastic iterated prisoner's dilemma".*

[22]. Laurene V Fausett. “*Fundamentals of Neural Networks: Architecture, Algorithms and Applications”*.

[23]. Martin T Hagan, Howard B. Demuth and Mark Hudson Beale. “*Neural Network Design”*.

[24]. Munakata, Yuko ; Pfaffly, Jason. “*Hebbian Learning and Development*”.

[25]. Wulfram Gerstner and Werner M. Kistner. “*Mathematical Formulations of Hebbian Learning*”.

[26]. Richard P Lippmann. “*An introduction to computing with neural nets*”.

VITA GRADUATE

COLLEGE

UNIVERSITY OF NEVADA, LAS VEGAS

SARA S PALLEKONDA

Degrees:

Bachelor of Technology in Computer Science and Engineering, 2009

Jawaharlal Nehru Technological University, India

Thesis title: Modeling of Cellular Automata and Agent-Based Complex Systems

Thesis Examination Committee:

Chairperson, Dr. Wolfgang Bein, Ph.D.

Committee Member, Dr. Lawrence Larmore , Ph.D.

Committee Member, Dr. Evangelos Yfantis, Ph.D.

Graduate College Representative, Dr. Muthukumar Venkatesan, Ph.D.