ON DEVELOPMENTAL FORMATION OF PATTERNS

Mo Alian

A thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy (Computer Engineering)
Concordia University
Montréal, Québec, Canada

April 2013

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Mo Alian**
Entitled:	**On Developmental Formation of Patterns**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr.	______	Chair
Dr. D. Precup	______	External Examiner
Dr. P. Grugono	______	Examiner
Dr. L. Varin	______	Examiner
Dr. A. Agrawal	______	Examiner
Dr. N. Karma	______	Supervisor

Approved ______

Chair of Department or Graduate Program Director

______ 20 ______ ______

Dr. Robin A.L. Drew, Dean

Faculty of Engineering and Computer Science

Abstract

On Developmental Formation of Patterns

Mo Alian, Ph.D.

Concordia University, 2013

The constantly increasing amount of resources available to engineers and scientists have allowed them to target larger problems whose size and complexity introduce new challenges: the time required to find the solution is longer, the solutions are more error prone, and the tests and repairs are more expensive. Self-organizing methods have recently been the promising pioneers in dependable robust design. Distributed self-organizing patterns can emerge to demonstrate the desired characteristics, either in form or functionality. At the same time, being inspired by the natural development of multicellular organisms, researchers have started using artificial development to improve features such as scalability or fault tolerance of the solution. However, the current solutions resulting from artificial development are either very small in size or very simple in architecture.

The first part of this thesis introduces a method to emerge patterns that demonstrate given functionality whose architecture is not known in advance. The notable achievement is the innovative fitness function in the evolutionary algorithm used there, which increases the density of the solutions in the search space and more importantly, makes the often-extremely-rough search space smoother.

The second and major part of this thesis studies formation of given large patterns from simpler initial patterns. This problem is solved in the framework of Cellular Automata. We push our methods to their limits by targeting large non-periodic patterns that have not been originally created by developmental methods. We use patterns for which the similar existing methods take a long time to find the solution, and their solutions are often large and seldom scalable. We suggest improvements to the existing methods to allow them find more efficient solutions, and also present two new methods to improve the results even further. In the end, we show that our suggested method also contributes to scalability. More specifically, our second suggested method decreases the growth rate of the solution to be slower than the growth rate of the problem size.

To Rezvan and Reza, my dearest parents who never stopped believing in me,
and to Jaleh, Ghazal and Bahareh, the best sisters one can ever have.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction and Background

1.1 Introduction

The improvements of the engineering tools and computational resources has made researchers eager to tackle larger and more complicated problems, for which large designs are a necessity. Such large designs enable researchers to target new problems in new scales and in a vast domain of applications. Large designs, however, come with large complications. Designs get more error prone, tests become more difficult, and maintenance becomes more expensive as the size grows. Finding, storing and running algorithms become complicated. Applying even minor modifications to the design in order to adapt to a slightly different problem can become extremely expensive when the size and complexity grow. It is not a surprise anymore if the costs of test and maintenance of modern products exceed the cost of the resources required for the product itself. One can find many examples where discarding a whole faulty product and replacing it with a fresh one costs less than performing diagnostics and doing repairs.

On the other hand, natural designs do not seem to suffer from such complications. This is while the most sophisticated artificial designs still look astonishingly simple when compared to natural designs. Mimicking even small parts of the behavior of a natural design in a controlled environment is considered to be an important achievement for engineers. Even the simplest life forms have features such as adaptability, fault tolerance, or scalability that has long been a major target for the artificial designs. Expectedly, there has been a trend in artificial design to follow the fundamentals of natural designs to deliver such features to the artificial products.

Two simple yet fundamental elements that contribute to the robustness of natural designs are modularity and self organization. Modularity makes it possible to find relatively simple designs that can be added together to form complex architectures or perform complex behaviors. Modularity also contributes to fault tolerance in a way that only the faulty module - and not the whole product - gets replaced by another modules. Moreover, in the case of re-programmable modules, the faulty module can be corrected by reloading the module configuration from a healthy module. Modularity contributes to scalability and flexibility where similar modules can be added to the existing architecture to address a larger use case or a slightly different one. Self organization contributes tremendously to the fault tolerance of the design by removing the single points of failure. There are always chances for a centrally organized system to fail if the central organizer module fails. Self organization also dissolves the necessity of complex central algorithms for which expensive resources are usually required. Avoiding complex algorithms results in easier testing and maintenance which contribute to the efficiency of the design.

In this thesis we study the performance of self organizing, locally connected modules to emerging global behaviors or architectures for two different types of problems. In the first problem we generate patterns that demonstrate certain given functionalities while there is no restriction on the architecture. In the second problem we generate much larger patterns - for which the precise architecture is given - from simple initial patterns. Both problems use two dimensional grids of homogeneous modules with limited local connections. To demonstrate the problem and the solution intuitively we use two common applications, each of which used for one of the above-mentioned problems. The first problem deals with finding feed-forward gate-level digital circuits to implement given binary functions, and the second deals with finding descriptions of given gray-scale 8-bit images so that they can be re-generated on a grid of self organizing locally connected modules. Although they target different applications, both problems are cases of a bigger problem, i.e. developmental formation of patterns. Although we only study two dimensional patterns, the methods are easily extendable to patterns of higher dimensions. Each problem is defined in details in its own chapter.

1.1.1 Forming Patterns of Known Functionality

There are certain pattern formation problems when the concern is the global emerged behavior of the pattern. In other words, there is no restriction on the architecture of the pattern as long as the specific functionality has emerged. An example is digital circuits where only the interface is defined as the mapping between the input and output, and the internal architecture of the circuit is to be found.

In the first section, we present a new method to find developmental descriptions for gate-level feed forward combinatorial circuits. In contrast to the traditional description of FPGA circuits in which an external bit stream explicitly describes the circuit (including the internal architecture and the connections), developmental descriptions form the circuit by synchronously running an identical developmental program in each building block of the circuit. Unlike some previous works, the connections are all local here. Evolution is used to find the developmental code for the given problem. We use an innovative fitness function to increase the performance of evolution in search for the solutions. We also relax the position and order of the inputs and output(s) of the circuit to increase the density of the solutions in the search space. The results show that the chance of finding a solution can be increased up to 375% compared to the use of traditional fitness function. We show that this method is capable of describing basic circuits and is easily scalable for modular circuits.

1.1.2 Forming Large Patterns of Known Architecture

The other problem in pattern formation is the case where the precise architecture is given and the task is to find the simple local rules that starting from a simple configuration (i.e. a configuration with small amount of information) that once executed on the basic modules of the pattern, the complex configuration can be formed. The found local rules along with the simple configuration can be stored to generate the complex configuration. The reasons for being interested in such descriptions - instead of storing the final detailed configuration - is often to benefit from the products of developmental descriptions, mainly scalability and fault tolerance.

In the second chapter we present methods to find local rules that form gray scale, 256-level images from only the most significant bit of each pixel in the image. Cellular Automata is used as the framework to achieve this goal.

Despite its well suited properties for generating patterns, Cellular Automata has been mainly studied for its dynamic behavior and has been usually applied to solve dynamic problems. There are few known algorithms to solve the inverse problem of Cellular Automata; i.e. given a final configuration, what rules take the Cellular Automata from certain initial configuration to that final configuration. Chapter 3 offers improvements on the existing methods and also proposes several new methods for this task. The comparison of the results show that the methods represented in this thesis can outperform the existing methods in terms of taking both shorter time and less memory to find and store the solution.

1.2 Background

1.2.1 Evolvable Hardware Design and Developmental Description of Digital Circuits

Computer Automated Design has been successful for simple and novel artifacts but it usually faces difficulties when it comes to more complex designs. Scalability of the design can be a solution to the complex designs problem[58]. Hornby [31] states that by employing regularity, modularity and hierarchy in the design we can empower evolution to find larger and more complex designs in the search space using the same computing power. Also Bentley [5] states that generative encoding (or embryogeny design, as he uses in his book) can reduce the search space in the evolutionary search because of its compact genome. Above that, it is also claimed to be capable of finding more complex designs in the solution space. On the other hand, he ensures that such developmental solutions are difficult to design and evolve, and will suffer from issues such as bloat, pleiotropy and disruption of child solutions if not designed very carefully.

Evolvable Hardware Design (EHW) uses Evolutionary Algorithms (EAs) to find an optimum design of digital circuits in terms of surface, speed and fault tolerance. They can also use the physical characteristics of the underlying chip to improve its performance [60] [61] [63]. Miller [44] [41] [42] showed that EHW is also capable of finding innovative designs which outperform the traditional human design in terms of used resources. While EHW can address issues like efficient surface usage, fault tolerance and innovation, they suffer from an instinctively drawback of Evolutionary Algorithms: the solution is usually not scalable. This means that having the solution to the problem of the smaller size usually does not help to find the solution to the problem of the bigger size any faster. Instead, the runtime of the EA usually exponentially grows by the linear increase of the problem size. A solution to overcome the scalability issue in EAs is to break the direct mapping between the genotype and the phenotype. If the genotype has a one-to-one mapping to the phenotype, searching for more complex individuals will be equal to searching a larger and probably higher dimensional space. This eventually will make the EAs to fail finding the solutions to the large problems unless there exists a very efficient encoding. Developmental Programs that grow into a final circuit do not have this problem. The size of the circuit is not bounded by the size of the developmental program (DP), and it is possible to have one DP growing into fully functional circuits of vastly different sizes. In approaches like CGP [43] [40], although the solution is a developmental code which defines the connections between the cells but still needs an external module to do the routing between cells on a physical configurable circuit.

Gordon and Bentley in [7] define the external, explicit and implicit embryogenies. The growth process is fixed and external to the genome in the first type, and only the parameters are optimized. In the explicit embryogeny, the growth rules are optimized in the same way a program is evolved in genetic programming. The last one includes a highly interactive chain of rules which are running in parallel and affect each other to develop the phenotype. The implicit embryogeny is closest to the biological process of development and is claimed to be the most successful type of embryogeny in scalable designs. The same authors introduce two implicit embryogeny models to find 2-bit adders in 2 by 5 CLB FPGAs [22]. Neither can find a perfect solution but the close-to-optimum solutions exhibit good scalability and cell differentiation. They conclude that developmental evolution in that experiment is outperformed by naive genome representation. To understand the reason of failure in that experiment, they hand designed a functional 2-bit adder in a circuit composed of 2 by 5 cells [21], and found its parameters (which were named *proteins* there). By studying the results, they noticed that the lack of cellular communication and low resolution has been preventing them to find the optimum solution. The lack of communication existed because each cell detected the proteins only if it is created by the majority of the surrounding cells, and did not detect proteins created by specific neighbor cell. Also it had only two levels for amount of each protein: either on or off.

Evolution however failed to find an answer to their problem even after the above issues was fixed. They had to try different circuit size as well as different evolutionary parameters to make it work. This shows that despite all the advantages, the design even for a 2-bit adder is a non-trivial task. Gordon states that because of the cells protein sensing mechanism and the D4 symmetry of the circuit structure, the design has a strong bias to be D4 symmetrical [23]. He lists the symmetrical circuits, the not *large and highly irregular* circuits and the circuits which do not use absolute but relative positional information as the ideal candidates for this approach. Another advantage of this method other than finding the pattern is that it maintains the found pattern, i.e. the design can return to its original configuration after small perturbation in the design.

Stanley et. al. [59] show that they can achieve *reuse* of the modules by copying or updating genome parts, but this type of reuse leaded to inflexible structures. Their approach was different from the nature's approach toward reuse. Nature evolves repeating elements simultaneously based on its bilateral rules and not by discovering parts separately and then combining them together. It was concluded that their conventions and encoding has not been the best choice for their designs. The encoding can be of extreme importance when looking for a scalable design, because *the characteristics of evolved design are limited and biased by the representation* [32]. Hornby then focuses on the representation for evolutionary algorithm and look for the possible properties for design representation. By comparing Angelines classification [4] (translative, generative and adaptive) of the representation with of Bently and Kumars [7] he suggests 3 properties for the evolutionary design to address the scalability: Hierarchy, Modularity and Regularity. Hierarchy is done by combination, and creating more powerful expressions from simpler ones. Modularity is done by abstraction, for example using labeled procedures and functions instead of using only low level instructions. Although scaling up a developmental design is expected to help us solving more complex problems, we need to know what parameters are scalable. Tufte [64] lists 3 domains of developmental design which can be scaled:

- Phenotypic resources: number of cells available for development to exploit
- Developmental resources: number of possible cell types (functions of a cell), number of developmental steps, etc.
- Computational resources: number of neighbors or cell states, number of clock pulses to flip flops in a cell, etc.

He also provides some results to show that scaling the above resources can impact the scale of the design to solve bigger and more complex problems. While he shows that regularity in the design can lead the search to find more complex problems, Hartmann et. al. [26] shows the other way around. They claim that regularity in a functional digital circuit is more than random circuits. Their scale for regularity is the compressibility of the description of the circuit, and they claim that the compressibility is higher in the bit string describing a functional digital circuit than a random bit string. This argument however seems to be very dependant on the representation of the circuit, and does not sound very convincing. There have been researches other that scalability of evolvable hardware studying the robustness of the design. Sipper et. al. [52] show the ability of replication and re-generation of the systems (usually automata in their work) in ontogenetic design. This ability leads to self-repair and also creation of identical organism by duplicating the genome of the zygote to another cell. As examples of ontogenetic systems, he lists the following researches:

- Von Neumanns self replicating automata has universal computing power, universal construction power (i.e. it can construct any automaton) but it is so complex that it did not have any physical implementation until 2000. [9]
- Langtons self replicating automata is much simpler but it has only self-replicating power, with no computational power.
- Tempestis self replicating automata has finite or universal computational power.

- Manges embryonics is a Cellular Automata, capable of universal computation and simple enough for physical implementation.

In Chapter 2 we present a method to implement any combinatorial digital circuit in gate level on a grid of configurable hardware elements. The main contribution of this work is that the resulting circuit includes sufficient information to build the functional circuit, including the gate arrangement and the routings. Keeping in mind that a considerable amount of resources on the configurable hardware (e.g. FPGAs) and the circuit compilation time is dedicated to the routing and connections, this property of our method tends to be attractive for practical problems. Also we try to improve the traditional fitness function used in EHW (for example the fitness function used in [42] and [[27] or the basic component of the fitness function in [15]) to move toward the optimum solution more efficiently.

1.2.2 Pattern Formation in Cellular Automata

Cellular Automata (CA) is a grid of cells with local connections and limited number of valid states. The state of the cells are updated synchronously and in discrete time steps. The state of each cell at each time step is determined according to the state of its surrounding cells at the previous time step. It was first introduced by the computer scientist Janson Von Neumann, and the mathematician Stanislaw Ulam in 1940s. It was originally of interest of those researchers because of its simple model and its *self replication* feature when placed in a suitable environment. The problem Von Neumann was trying to solve was finding the logical organization sufficient for an automaton to make a copy of itself [65].

CA is known for its potentials to perform relatively complicated global tasks using simple local rules. The tasks passed to the CA usually involve encoding the input in a form of initial CA configuration (i.e. the combination of the states of all the cells) and letting the CA to run either for a certain number of steps or until it reaches a stable state, where there is no further changes in the states of the cells. At this point the configuration of the CA is decoded and is interpreted as the answer of the CA to the given input. The main challenge will be designing or finding the rules that given the initial configuration of the CA can generate the final configuration.

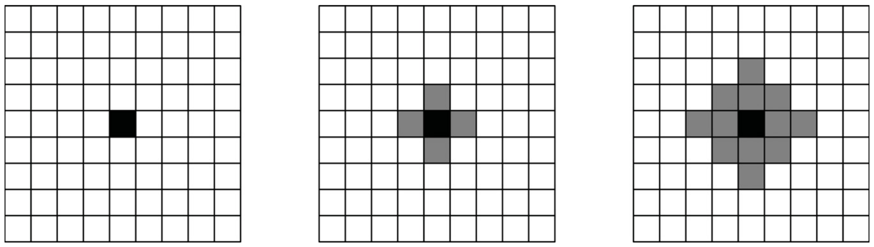


Figure 1.1: 2-D Von Neumann neighborhoods of radius $r = 0$ (left), $r = 1$ (middle) and $r = 2$ (right)

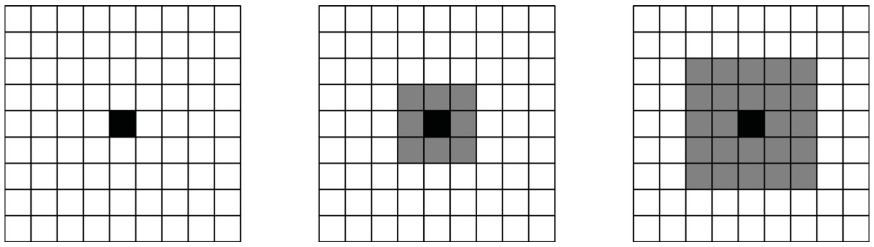


Figure 1.2: 2-D Moore neighborhoods of radius $r = 0$ (left), $r = 1$ (middle) and $r = 2$ (right)

CAs mentioned in the literature often own a one or two dimensional structure where each cell has a binary state. In more general cases, the cells can posses a state from a larger reference set. It is also possible to have an n-dimensional CA as explained in [69]. The neighborhood of a cell can be either Von Neumann neighborhood with radius r (Figure 1.1), Moore neighborhood with radius r (Figure 1.2) or a free form neighborhood (Figure 1.3) [70]. The cell whose neighborhood is demonstrated is colored in black, and the neighborhood is colored in gray.

One can divide the CAs into two types from an applicational perspective: *problem solver CAs* [24] and *pattern generator CAs*. Table 1.1 lists the main differences between the two types.

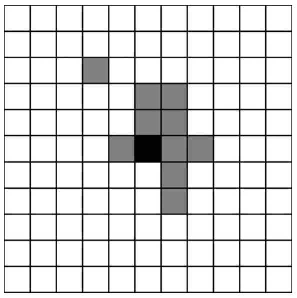


Figure 1.3: An example of a free-form 2-D neighborhood

Table 1.1: Comparison of the problem solver and pattern generator CAs

CA type	Problem Solver	Pattern Generator
Number or valid initial states	*Many*	*One*
Number of acceptable final configurations	*Many*	*One*
Input and Output format	*Encoded*	*Raw*
Final Configuration	*Unknown*	*Known*
Basis of validation	*Outputs*	*Configuration*

The *problem solver CAs* are used to solve a wide range of problems where every and each of the valid inputs should map to a correct output after finite number of steps. Both inputs and outputs can have a many-to-one mapping to the CA configuration. In other words, a specific input can be encoded with more than one CA configuration and the CA should produce a valid correct output disregarding which of the many possible configurations the input has been encoded to. Similar for the output, more than one CA configuration can be decoded to the same output. We call this set the *Computational Problems*.

It has been shown that some versions of CA have the universal computational properties equivalent to the Touring Machine. One of the most famous examples of the computational powers of CA is Conway Game of Life [8], were each cell in a 2D CA is either in the *on* or *off* state. Complicated entities and interactions between them can emerge from a constant set of simple local rules and appropriate initial configuration, eventually forming concepts such as message transmission or performing logical and mathematical functions.

Another type of the computational problems to be solved by CA are the synchronization problems such as the famous *Firing Squad* problem [45]. This problem is defined as follows:

> At time step 0, all cells are in the quiescent state. At some time step $t = t_i$, the general (responding to an external input) goes into a special state, interpreted as a *command to fire*. Then at some later time step $t = t_f$, all of the soldier cells must go into the *firing* state, and none of them can have been in the firing state at any previous time step. The problem is to devise states and state transitions for the soldiers that will accomplish this behavior. [24].

In the original CA used for this problem, they needed 15 different states for the cells but there has been a comprehensive research on finding the optimum or close-to-optimum number of states and time steps needed to synchronize the firing squad [38] [39].

Another example of the computational problems is the parallel formal-language recognition, in which a finite string of characters should be decided if belongs to a formal language or not [54], where it is proved that the class of languages that can be accepted by Binary CAs is the class of context-sensitive languages. Cellular Automata has also been used for image processing. Rosin in [48] finds rules of the Cellular Automata that can perform tasks such as noise removal, thinning and convex hulls. Another example is to perform logic or mathematical functions on binary inputs, where the combination of two or more input is encoded into an initial state and the CA will produce the configuration that decodes to the correct answer [16].

On the other side there are the *pattern generator* CAs, those who do not accept encoded inputs and are not expected to provide encoded outputs. A CA of this type is designed to start from a certain initial configuration and is expected to emerge to a pre-designed configuration (i.e. pattern). The final pattern can be either a steady configuration that emerges after a finite number of time steps, or a transient configuration at a certain time.

Unlike the research on computational aspects of CA - which goes back to 1940s - the research on using CA and its variants for pattern formation did not start before 1990s. Kauffman [34] had one of the first studies on applying evolution to find self-organizing patterns in CA, and the idea was followed by emerging studies such as modeling the self-assembly processes of electro-statics [56], testing and growing digital circuits [13], growing patterns [47] and modeling the developmental process [35] [6]. Ozturkery [46] suggests a method to form stable patterns (either random or regular) on a 2D CA. The unique feature is his method is that each cell updates itself and all of its 8 adjacent neighbors, and cells get updated not simultaneously but sequentially one after the other. Also Yang and Billings [72] use genetic algorithm to find both 1D and 2D CAs. Although they find both rules and the neighborhood, the rules are very simple and the neighborhoods contain the adjacent cells only. As we will see in Chapter 3, we try to solve that problem for any set of rules and any neighborhood. Also unlike their methods, the patterns used in this work are not a result of running a certain rule on an initial pattern. Instead, we try to suggest methods that work for any arbitrary pattern.

The size of the CA in the works we have seen so far has been very limited, usually much less than 100 cells. Despite the strong computational abilities of CA, it has been observed that controlling their behavior and making them to emerge specific patterns are extremely hard. This is believed to be due to the extreme sensitivity of CA to small changes in the initial condition or the rule set. The second problem to be studied in this thesis is generating large patterns on CA, where the CA contains at least 500 cells.

1.3 Problem Statement

In the first part of this thesis we deal with the patterns that accept inputs and provide outputs to such inputs. We present a method to find the developmental description that generates a pattern, which maps the given inputs to appropriate outputs. We chose digital gate-level feed forward combinatorial circuits as our application for this. More specifically, we solve the problem of finding a combinatorial gate-level digital circuit with m inputs and n outputs and the given mapping between the inputs and the outputs. The task of our method is to find the developmental program that once executed on a specific sized circuit for a known number of time steps, a circuit with the given functionality emerges from a simple, well-defined initial state. The developmental program, size of the circuit, the number of time steps and the simple initial state are to be found by our method.

There is no input or output defined for the patterns that we deal with in the second part of the thesis. There we study the formation of patterns from simple initial configurations, where only the form of the pattern is important. The form of the pattern is already known in this part and we solve the problem of finding the developmental description that generates the given complex pattern from a simple initial pattern. This way one can store the simple initial configuration and the developmental description to generate the more complex pattern.

We evaluate the success of our methods in several measures and comparing them with existing methods. Among those measures we can name evolvability (the success of evolutionary algorithms in finding the solution), memory efficiency (the amount of memory required to store the developmental descriptions), algorithm complexity (time required to find the solutions) and scalability (the ratio of the growth rate of the problem size to the growth rate of the solution size). Our objective is to improve these measures in our methods comparing to of existing methods.

1.4 Thesis Contribution

We provide a method to emerge the patterns that demonstrate the given functionality in Chapter 2. The most important contribution of this chapter is the innovative fitness function explained in Section 2.2.3. That fitness function increases the density of the solutions in the search space and more importantly smooths the space of solutions. In our experiments, this resulted in 375% increase in the chance of evolution to find a solution for the given problem.

In Section 3.2.1.2 we improve the efficiency of the existing methods for solving the inverse problem of Cellular Automata. We reduce the amount of memory required to store the transition function by 8.36% on average for all CAs, and by 12.65% for memoryless CAs in our experiments. These results are provided in Section 3.3.3.11. We suggest two band new methods to solve the inverse problem by adding *hidden states* to the Cellular Automata in Sections 3.2.2 and 3.2.3. This reduces both time and memory requirement of the transition function. The first method reduces the memory requirements for storing the transition function by 3.31% on average for all the CAs and by 39.5% on average for the memoryless CAs (Section 3.3.5.1). The calculation time is decreased to only 4.84% of the shortest version of the existing method. In Section 3.3.8 we will see that the best variation of our second suggested method requires slightly larger memory to store its discovered transition function comparing to our first suggested methods (0.38% larger on average) and takes slightly longer time, but as we will explain, it is more scalable than both existing methods and our first suggested method.

The scalability of each methods is measured and analyzed in Section 3.3.9. We will show that our suggested methods are more scalable than the current results, at least for our test data. We will see that while the rate of growth of the memory required to store the transition function is faster then the rate of the growth of input in the existing method, the results of our first suggested grow almost as fast as the input size, and our second method grows even slower than the input size.

1.5 Thesis Organization

We target the problem of *Forming Patterns of Known Functionality* in Chapter 2. We introduce the framework and define the problem to be solved in Section 2.1 and present our method to solve it in Section 2.2. The results and their analysis are provided in 2.3.

Chapter 3 forms the major part of this thesis. We review the cellulat automata in Section 3.1.1 and its applications in Section 3.1.3. We review the current existing methods to solve the inverse problem of Cellular Automata and suggest our improvements in Section 3.2.1. We present our first new method to solve the problem in Section 3.2.2 and our second new method in Section 3.2.3. Sections 3.2.3.2 to 3.2.3.6 explain the variation of our second method along with the analysis of each variation. We evaluate and analyze the results of our improvements to the existing method, and both our new methods in Section 3.3, with a note on scalability of our methods in Section 3.3.9. The summary of this thesis is presented in Chapter 4.

Chapter 2

Using Developmental Encoding to Emerge Given Functionalities

In this chapter we propose a method to generate patterns on grids of homogeneous cells with defined local connection, while the architecture of the solution is not known in advance. As we mentioned in Chapter 1, the majority of existing methods solve the problem from the perspective of a global designer, a unit who has full access and complete control over each cell in the grid. In contrast, we use self organizing cells and find developmental programs that emerge the given global functionality once they are ran synchronously on all the cells for a certain number of time steps.

A functionality is defined as the ability to generate outputs from the given inputs according to explicit logic functions. The unique characteristic of this method is the freedom of the algorithm to choose the the location of the inputs and the outputs on the pattern. This gives the algorithm a higher chance to succeed than the methods who fix the location of the inputs and outputs. We will show that as expected for a developmental formation, this method exhibits both scalability and fault tolerance, two important properties of robust design.

To give our method a realistic touch, we define our application in the field of digital design. On a network of locally connected cells each of which can be configured to be one of the many possible simple 2-variable binary functions, our goal is to find the developmental program that is ran in all the cells simultaneously to let the cell chose its inputs and function. The combination of the inputs and functions of the cells enables the global circuit to perform the given multi-input multi-output binary function. The location to read the global inputs and the location to write the global output as well as the number of time steps to run the program is determined by the same algorithm.

We define the problem in details and explain the structure of our framework in Section 2.1 and present the algorithm to find the solution in Section 2.2. Section 2.3 provides results for test problems and Section 2.4 provides the conclusion and prepares us to switch to the other case of generating patterns, fully explained in Chapter 3.

2.1 Problem Definition

The problem to be solved in this chapter is to design a framework to find developmental programs that develop feed-forward gate-level digital circuits that implement given binary functions. Such programs start development from a simple homogeneous configuration of a certain size. The main challenge will be finding such developmental program that develops a non-previously-known architecture which exhibits the given functionality. The inputs to the problem are therefore the number of inputs, number of outputs and the mapping between them (e.g. in form of a truth table). The output of our method will be a developmental program, and the certain number of steps to let that developmental program run on an initially clear configuration.

2.1.1 The Framework

A circuit in our method is a two dimensional array of configurable cells. The inputs to the circuit are provided through the left-most cells and the outputs are read from the rightmost cells. This means that the direction of the signals is from left to right in a high level abstract view (Figure 2.1). To implement this, each cell $c_{i,j}$ (a cell in row i and column j of the circuit) can only accept inputs from either $c_{i-1,j-1}$, $c_{i,j-1}$ or $c_{i+1,j-1}$ (Figure 2.2). Such limitation on the connections lets the circuit form without the need of any external processing module for the routings, as in Cartesian Genetic Programming (CGP) [43]. In

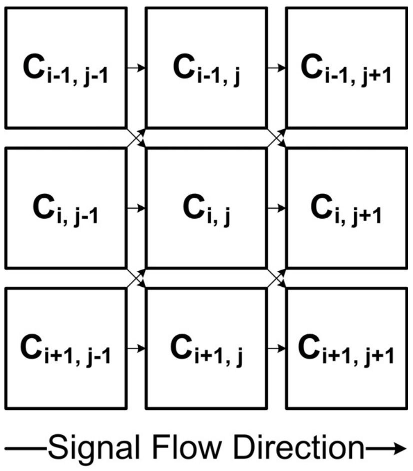


Figure 2.1: The grid of cells in a circuit

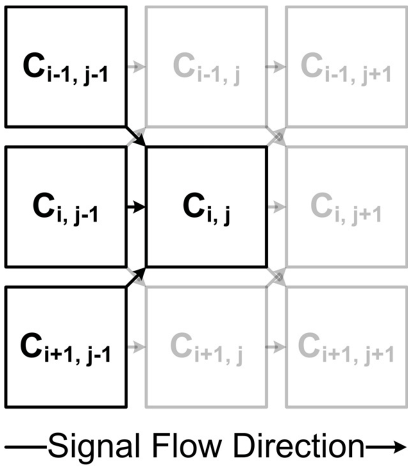


Figure 2.2: Potential inputs to a cell

CGP the cells form a one dimensional array and each cell c_m can be connected to any cell c_n as long as $n < m$. While that condition lets equivalent circuits to ours to be formed, it needs to have a routing mechanism for the circuit to physically connect the cell inputs to the other cell outputs. The circuit resulting from our method do not have such a demand. This means that once each cell sets its own function and input connection to the adjacent cells, the routing is already done and there will be no need for any external routing mechanism.

Each cell in the circuit has an identical developmental program in form of set of rules, and also 5 properties, each taking an integer for their value. Figure 2.4 represents an abstract view of the cell. Being inspired by the CPU architecture and its OpCodes, *GPR* stands for *General Purpose Register*. This name was chosen because this property is available to be

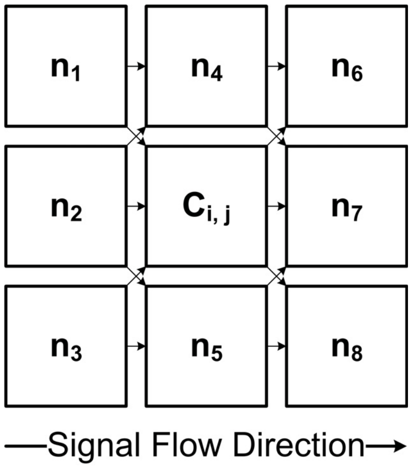


Figure 2.3: Naming conventions of the neighbors

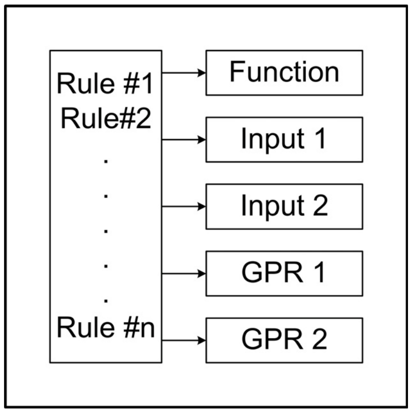


Figure 2.4: Abstract of the architecture of a cell in the grid

modified and used in any way the developmental program wants. The cells at the borders of the circuit are named border cells and all their properties are permanently set to -1. For all other cells, the initial values of all properties are 0. Table 2.1 lists the cell properties and their possible assigned values for non-border cells and table 2.2 lists the equivalent cell function for each value of the *function* property.

2.1.2 Developmental Program

The developmental program is stored in form of set of rules in what we call the a genome in this method. The circuit size is fixed to a certain size at the beginning, and there is no growth in terms of increasing the number of cells in the circuit. The genome is simply a variable number of ordered *IF-THEN* rules (Figure 2.5). The *IF* part can put any condition on any property of any neighborhood cell. Based on the values of that property, the rule can set or update any property of itself. The rule shown in Figure 2.5 is read follows:

> *IF* the property p of the neighbor n has the relation r to the value a
>
> *THEN* according to s, assign either the value a, b, *b+1* or *b-1* to the property p' of the cell.

In which p and p' can be any property of a cell (e.g. function, first input, etc), n is the index of the neighbor (0 to 7, for any of the 8 adjacent cells in Figure 2.3), r is one of the possible relation from Table 2.3 and a and b are the possible values for p and p', respectively. The list of possible actions on the parameter b is listed in Table 2.4. Only the parameters n, p, r, a, s, b, p' are stored in the genome. For example, the third rule in Table 2.5 ($1\ 0\ 1\ -1\ 3\ 0\ 0$) reads as follows:

> If the *function* of the neighbor 1 is equal to -1,
>
> then set the $GPR1$ property of the cell to 0.

There are 4 pre-written rules in the genome which affect the $GPR1$ and $GPR2$ properties of the cell. These rules are manually designed and added to the genome (Table 2.5). These rules aim to simulate the protein gradient along the embryo of multicellular organisms at

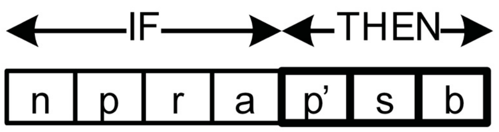


Figure 2.5: Structure of a single rule in the cell's rule base

Table 2.1: The cell properties and the range of valid Integer values of each

Parameter	Function	Input 1	Input 2	GPR 1	GPR 2
Range of Values	$[0,7]$	$[0,2]$	$[0,2]$	$\mathbb{Z}$	$\mathbb{Z}$

Table 2.2: Cell's output according to its *function*, *Input 1* (I_1) and *Input 2* (I_2) properties

Function Value	0	1	2	3	4	5	6	7
Cell's output	0	I_1	$\neg I_1$	$I_1 \wedge I_2$	$I_1 \vee I_2$	$I_1 \oplus I_2$	$\neg(I_1 \oplus I_2)$	$\neg(I_1 \wedge I_2)$

Table 2.3: Possible values of r in a rule and their corresponding relations

Value of r in the rule	0	1	2	3
Interpreted relation	$=$	$\neq$	$<$	$>$

Table 2.4: Possible values of s in a rule and their corresponding actions

Value of s in the rule	0, 1, 2	3	4	5
Interpreted action	Assign b	Assign a	Assign $a+1$	Assign $a-1$

Table 2.5: The 4 pre-defined rules in the genome

Rule Index	Rule
0	1 3 3 -2 3 4 0
1	3 4 3 -2 4 4 0
2	1 0 1 -1 3 0 0
3	3 0 1 -1 4 0 0

the axis specification step [20]. The rest of the rules in the genome are generated randomly using an even distribution random generator, and are tuned during the course of evolution. The number of rules in a genome is limited to 25 plus the 4 pre-written rules, a total of 29 rules. We need to remind that similar to any rule in the genome, the pre-designed rules are prone to evolution as well. It is up to evolution to keep them or modify them in any manner that contributes to the fitness of the individual. During the development of the circuit, cells update their structure synchronously. A developmental step is composed of updating all the columns of the circuit, starting from the leftmost column and moving to the next column at the right until reaching the rightmost column. Updating each column is done by updating the topmost cell in the column and then move to the next cell at the bottom, until reaching the lowest cell in the column. A solution is a genome (i.e. rule base) which leads the desired behavior to emerge in the circuit after going through a certain number of developmental steps. The number of developmental steps needed for this is determined by the evolution, as is the genome itself.

2.2 Methodology

2.2.1 User Interface and Problem Statement

We use evolution to find the solution to the given circuit design problem. As explained in Section 2.1, the solution is a developmental program of the format mentioned in Section 2.1.2, necessary number of steps for the circuit development, and the size of the circuit. Note that the developmental program itself does not provide or care about the size of the circuit. Any developmental program can be run on any circuit of any size. It is evolution's job to find the appropriate circuit size for the developmental program.

To define a specific problem user has to state the number of inputs, number of outputs, and the mapping between the input patterns and the output(s). The latter one is done by telling the program the set of minterms created on each output pin. No information about the circuit's possible internal architecture is provided from the user. For example, Equation 2.1 defines a full adder.

$$\begin{aligned}
&Number of inputs : 3\\
&Number of outputs : 2\\
&output[0] = 3, 5, 6, 7 (carry)\\
&output[1] = 1, 2, 4, 7 (sum)
\end{aligned} \tag{2.1}$$

Evolution also gives the exact position of each input and output signal on the circuit. Unlike previous works in which user had to fix the position and the order of input and output signals, evolution is free to find the optimum placement of the I/O signals on the circuit. It is easy to realize that relaxing the I/O interface in this manner increases the density of the solutions in the search space. The easiest support of this is that the horizontal flip of a solution circuit is now a solution circuit itself, something which will not be the case if the inputs are fixed. The inputs are always provided on the left border ($c_{i,0}$) and the outputs are read from the right border of the circuit ($c_{i,N-1}$, where N is the number of columns in the grid). Figure 2.7 shows a sample full adder found by the program for the above description. It is important to remember that the program does not directly find the circuit in that figure, but a generative code which makes the circuit after going through the developmental process.

2.2.2 Evolutionary Algorithm

Evolution starts by creating a fixed sized population of random individuals. The population size was 500 in most of our experiments. As explained in [25], for evolution of cooperative rule base systems for static problems in which all the training instances are available, the Pittsburgh approach [55] with the individual having the whole rule-base works better than the Michigan approach [30] in which each individual is only one rule and the whole population together form the rule-base system. Each individual then is a complete circuit here, including the developmental program, the number of developmental steps for that program and the size of the circuit. To create a random individual first we create a random sized circuit with the restrictions of Equation 2.2.

$$\begin{aligned} Number of inputs + 2 \leq & Number of rows \leq 2 \times Number of inputs + 2 \\ 3 \leq & Number of columns \leq 2 \times Number of inputs + 3 \end{aligned} \tag{2.2}$$

This is because the minimum number of rows needed to provide the inputs is equal to the number of inputs and we also need two rows for border cells. The minimum number of columns is one column for the functional cells plus two columns for the border cells. The higher limit for the number of rows and columns is limited in regard to available processing power. The number of rows and columns in the initial population are evenly distributed between the two limits. After setting the size of a circuit, a random genome is created and assigned to the circuit. The random genome is composed of a random number of rules (limited to 29), with the first 4 pre-designed rules listed in the Table 2.5. The rest of the rules are filled with evenly distributed random parameters.

The fitness function is one of the main contributions of this work and is described in details in Section 2.2.3. We use fixed population size with 2% elitism. Parent selection is a tournament selection of size 3, and each parent goes through either mutation or cross-over with another parent to create the new individuals.

The main reason that tournament parent selection is used is to maintain the diversity of the population. Diversity can be measured in both genome and phenome level. When fitness proportional parent selection used, both genomic and phenomic diversities drop quickly. Figure 2.6 shows both diversities during one run of the evolutionary algorithm for the full adder. It is observed that the genomic diversity maintains its value much higher than the phenomic diversity. The reason is believed to be that redundant rules in the genome which never get activated. If two genomes have the same effective rules and different redundant rules, they will be translated to the same phenome while the genomes differ. The redundant

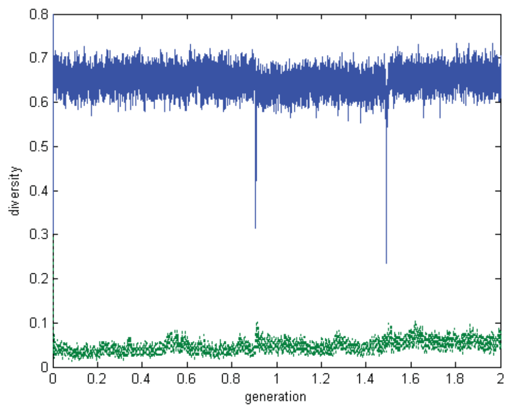


Figure 2.6: The genomic and phenomic diversity in one run of evolutionary algorithm

rules can be removed once the solution is found. That will make the implementation of the physical circuit cheaper and more efficient.

The ratio of mutation to cross-over is 1 in this experiment. There are 4 types of mutation which either add, delete or swap rules, or change the parameters of a rule in the genome. The different mutation types have equal chance to happen, and so is the chance of each parameter to be changed in the parameter level mutation. Cross-over can be either single point or shuffle and is always in rule level. Each individual on average creates two children and adds them to the intermediate pool, which is then sorted and the rest 98% of the next generation population are selected trough the fitness proportional selection. It is important to remember that the fitness of a circuit is only a measure of the behavior of the circuit (i.e. its response to the provided inputs) and reflects neither the circuit size nor the genome size. The effect of the circuit and genome size is in the tournament parent selection. If the two randomly selected individuals have the same fitness in the parent selection step, the one with smaller circuit size is selected. If this does not break the tie, the one with smaller genome size is selected. Because the smallest possible size of the circuit or the genome are not known to the user, finding an individual with fitness equal to 1 is not enough to stop evolution. The evolution stops if a smaller solution with fitness equal to 1 is not found after a certain number of generations, or after 20000 iterations.

2.2.3 Fitness Function and Sensitivity Analysis

The fitness function is a critical and arguably the most important part of any evolutionary algorithm. It is the fitness function which forms the search space to be smooth or sharp, conducts the search in a more evolvable search space [73] and eventually guides evolution to the optimum solution. A good fitness function not only maximizes at the target solution point but also increases gradually as we get close to the solution. This however is not always a trivial task to pick up a suitable fitness function. The *distance* of two solutions is not always very clear, keeping in mind that we usually have no information of the structure of the optimum solution. We usually can rate the fitness based on the performance of individuals and not their structure. This makes our job a bit difficult in problems of digital circuit design.

Unlike the most of biological organisms in which a slight change in their DNA usually leads to a non-fatal and slight change in their functionality, a slight change in the developmental or non-developmental description of a digital circuit very often drastically change the behavior of the circuit. The reason for this is the crisp nature of the digital circuits in which changing one single gate might inverse the whole final outputs. An example of this is when an AND gate from which the final output is produced is replaced with a NAND gate. In the fitness function often used in EHW with a fitness scale between 0.0 to 1.0, such change will instantly drop the fitness from 1.0 to 0.0.

The fitness function often used in EHW is the number of correct outputs for all possible combinations of inputs [41] [27]. This method suffers from the issue mentioned above, i.e. a small change in the architecture of the circuit might drastically drop the fitness even if the architecture is very close to the correct architecture. This will form the search space to be extremely crisp with sharp changes, in which evolution needs to be very lucky to not miss the optimum solution. To help this, we have changed the fitness function to not only reflect the number of correct outputs to the combination of inputs, but also the sensitivity of the outputs to the change of each input. The effect of the latter part is most obvious in the given AND NAND example. While the AND and NAND gates have complementary outputs, their sensitivity to their corresponding inputs are the same. To explain this, consider a combination of the inputs in which the first input is 0 and the second is 1. The change of the first input from 0 to 1 will change the output in both gates (i.e. the gates are sensitive to the first input in this input combination). The change of the second input from 1 to 0 will not change the output in either of the gates, so they have the same sensitivity on both inputs. This holds true for all other combinations of inputs. Thus if the final output gate is changed from AND to NAND gate, our fitness function still rewards the individual. The traditional fitness function used so far does not consider this similarity and only takes the net outputs into the account. Note that our method only examines the sensitivity of the circuit to the primary inputs and not any intermediate signal.

Our experiments show that adding the sensitivity analysis to the circuits improves the efficiency of the search in term of decreasing the iterations needed to find the optimum solution. To support this, we tried 50 runs of the evolutionary algorithm with population set to 500 and for a maximum of 20000 generations to find a full adder. Using the traditional fitness function, evolution could find a solution in only 4 runs. Applying our described fitness function, this number was increased to 15. Improving the chance of finding a solution by 375 percent clearly shows the advantage of this new fitness function over the traditional for EHW. The other property of our method is that it sets development free to locate the inputs and outputs at any desired row. This is in contrast to other works that fix the position of the inputs and outputs and force the development to read the input from and make the input to those fixed positions. The fitness function here examines any possible combination of input and output positions and accepts the best combination as the input / output positions. The density of the solutions is thus increased in the search space.

2.3 Experiments and Results

We tried to find the developmental code for 3 different circuits including full adder, 2-bit multiplexer and 4-bit parity generator. Evolution was able to find the solution for all these circuits. Tables 2.6 to 2.8 present the found genomes, and Figures 2.7 to 2.9 illustrate the found solutions. The number of developmental steps for all shown circuits is 2.

Figure 2.10 illustrates the fitness of the fittest individual and also the average fitness in

Table 2.6: The genome to develop a binary full adder after 2 time steps on a 5×5 grid

Rule Index	Rule
0	6 1 3 0 0 2 4
1	4 0 2 3 0 4 7
2	5 2 0 1 1 3 0
3	3 4 3-2 4 4 0
4	3 4 1 0 1 5 1
5	4 1 1 1 0 1 5
6	0 0 2 5 1 0 1
7	0 4 2 1 2 3 1

Table 2.7: The genome to develop a 2-bit multiplexer after 2 time steps on a 5×5 grid

Rule Index	Rule
0	7 2 0 1 0 2 7
1	0 0 1 7 0 0 1
2	5 0 2 0 0 2 5
3	1 2 0 2 1 0 1

Table 2.8: The genome to develop a 4 bit parity generator after 2 time steps on a 6×4 grid

Rule Index	Rule
0	5 3 2 0 2 2 1
1	5 1 0 0 1 1 2
2	0 0 0 1 0 1 5
3	0 2 3 2 4 4 0

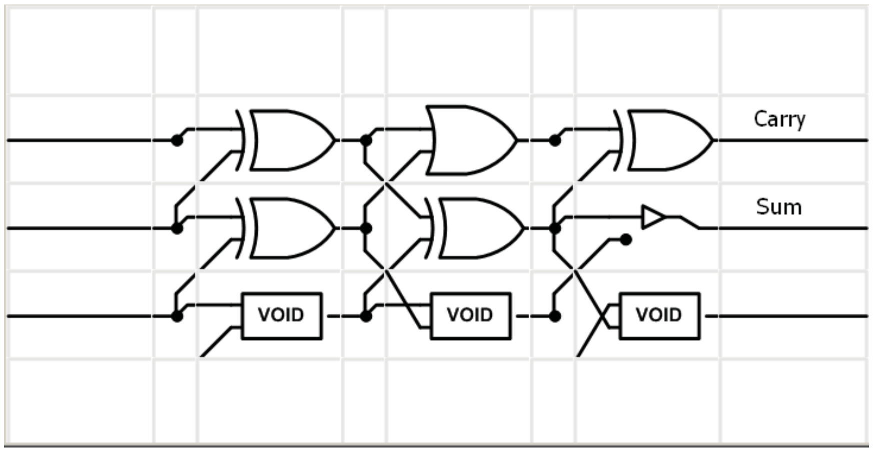


Figure 2.7: The full adder developed by the genome of Table 2.6.

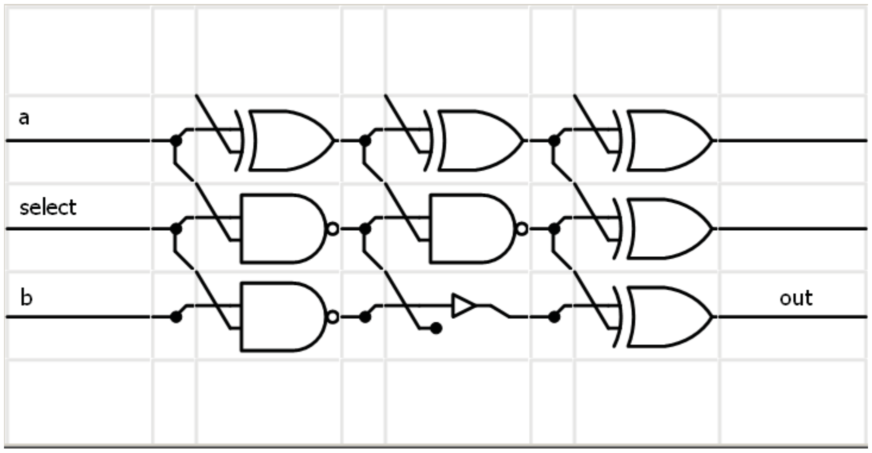


Figure 2.8: The 2-bit multiplexer developed by the genome of Table 2.7.

the population for the run evolutionary algorithm whose result was depicted in Table 2.6. We can observe from that figure that the average fitness in the population quickly follows the highest fitness in the population. This means that in each population there are lots of equally fit individuals. The two or three randomly selected parents therefore have a high chance to have the same fitness, leading the individual size to play an important role in parent selection. The experiments showed that considering the individual size in survival selection highly favors the small individuals and the EA eventually fails to find the desired circuit.

The scalability of the found solution for larger problem sizes was also studied. For this, first we ran the evolution 20 times to find 30 3-bit parity generators in a separate

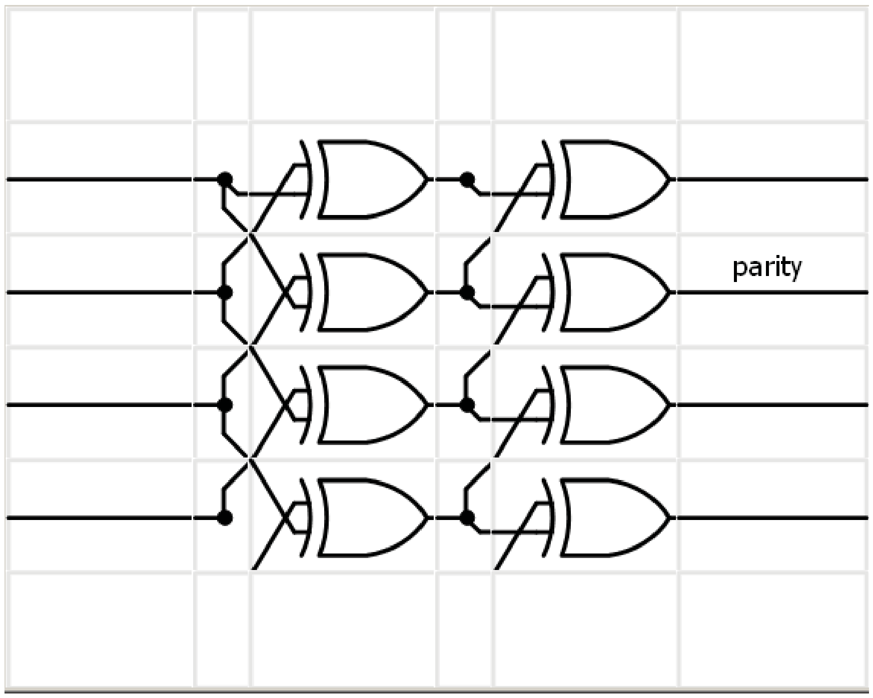


Figure 2.9: The 4-bit parity generator developed by the genome of Table 2.8.

experiment. Then we included those solutions in the initial population of the 4-bit parity generator problem, and ran the evolution 20 times to find 20 4-bit parity generators. We repeated the experiment 20 more times from the scratch, i.e. without including the solutions of the smaller sized problem in the initial population and measured the average of the best fitness in each generation. The results show that the performance of the evolution was incredibly increased when it included the solution to the smaller sized problem.

$$AdjustedFitness = k \times fitness + \frac{CS_{MAX} - CS}{CS_{MAX} - CS_{min}} + \frac{GS_{MAX} - GS}{GS_{MAX} - GS_{min}} \tag{2.3}$$

To compare the results of the two case studies we defined a new measure that includes all three parameters of circuit fitness, circuit size and genome size. The formula to adjust the fitness is shown in Equation 2.3, In Which k is a coefficient to enable the minimum increments of fitness to overcome the negative effect of resulting growth of the circuit and the genome size, and according to the fitness function explained in Section 2.2.3, as well as the implementation details is 128/3. CS and GS are the *Circuit Size* and *Genome*

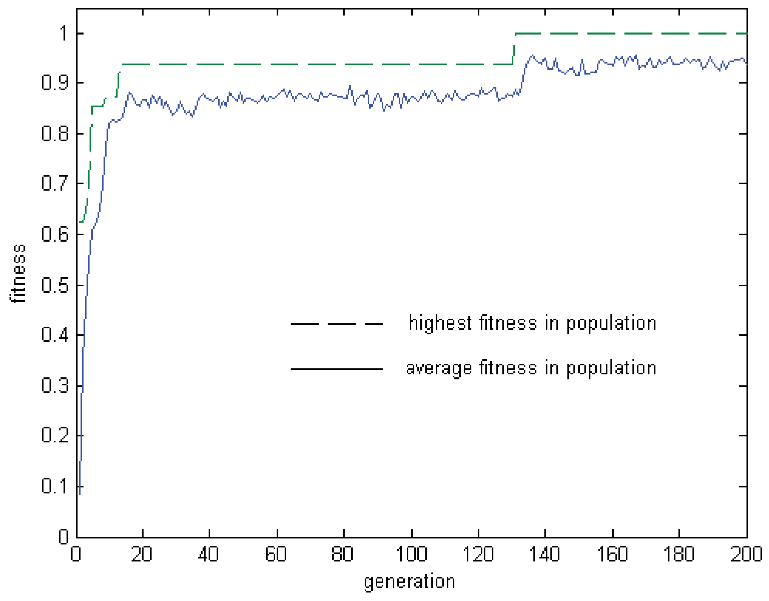


Figure 2.10: The highest and average fitness in the population for the evolution of the circuit in Figure 2.7.

Size of the current individual. CS_{MAX} and CS_{min} are the maximum and minimum of the *Circuit Size*, equal to 110 and 18 respectively (Equation 2.2) for a 4-bit parity generator. GS_{MAX} and GS_{min} are the maximum and minimum *Genome Size*, equal to 29 and 4 respectively (Section 2.1.2). Figure 2.11 shows the *normalized* performance of evolution when it already has the knowledge about the smaller sized solutions, compared to the performance of evolution when it starts from scratch - without any knowledge about the solutions of smaller sized problem.

2.4 Conclusion

The method presented here uses multiple properties per cell, including two general purpose registers. The amount of information available in these properties enables evolution to find developmental descriptions for the small digital circuits. Also , the sensitivity analysis introduced in thesis for the first time makes the evolutionary search considerable more successful. The found solutions also contribute to the scalability of the search for larger circuits of the same nature, as the population fitness has a noticeable improvement and the

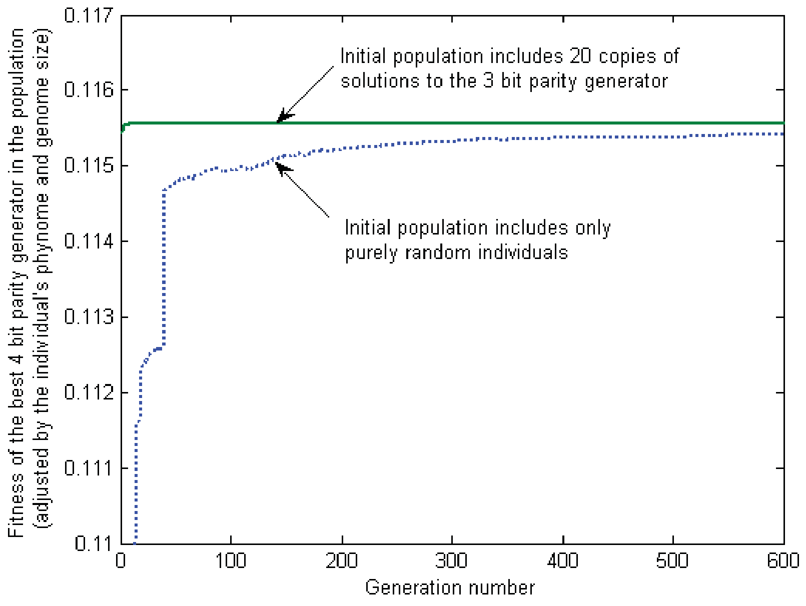


Figure 2.11: Improvement in evolutions performance by introducing the solutions of the smaller size problems in the initial population.

bigger solution is found in less generations if we already know the solution for the smaller size problem.

However, while the population is benefiting from containing the solution for the smaller problem, the evaluation of the individuals still takes an exponential time considering the size of individual. At the same time that the innovative fitness functions in this chapter increases the density of solutions in the search space, it also makes the evaluation time longer. PArt of the reason is that the evaluation of individual involves growing the solution first and then calculating the fitness. This requirement prevents us to examine larger patterns, where the growth can take a considerable time. For this reason, in the next chapter we study the pattern generation problem where the architecture is known in advance. That will let us to solve the pattern generation problem for considerable larger sizes, were there are at least twenty times more cells in the grid.

Chapter 3

Development of Large Patterns on Cellular Automata

3.1 Problem Definition

3.1.1 Cellular Automata

Cellular Automata is a grid of connected cells in a discrete space and discrete time, and is defined by the quadruple $\langle C, S, N, f \rangle$ where:

- C is a discrete connected *cellular space* of known dimension whose size can be either limited or unlimited in each dimension. Each individual cell in D-dimensional C can be represented as $c_{\vec{i}}$ where $\vec{i} = [i_1, i_2 \ldots, i_D]$. The CA can be either bounded or cyclic on any of its limited dimensions. A $D-$dimensional CA is cyclic on its limited dimension d ($d \leq D$) if:

$$\exists T \in \mathbb{N},\ \forall i_d \in \mathbb{I} : c_{[i_1,i_2,\ldots,i_d+T,\ldots,i_D]} = c_{[i_1,i_2,\ldots,i_d,\ldots,i_D]} \tag{3.1}$$

  Figure 3.1 shows a 2-D cellular space whose both dimensions are cyclic.

  The combination of the states of all the cells in C at a specific time t_k is called the *configuration* or state of the CA at time t_k. The union of all possible configuration of the CA is called the *State Space*. The number of cells in C is denoted as ς throughout

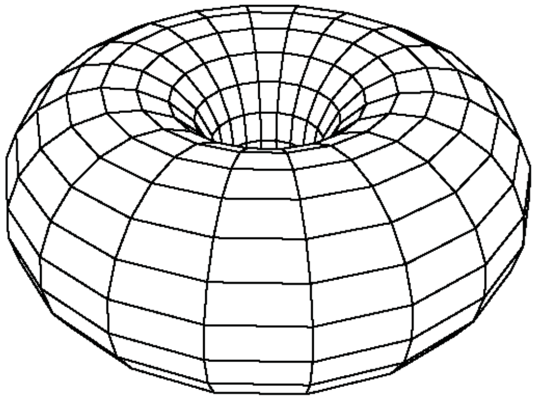


Figure 3.1: A 2-D cellular space with cyclic dimensions.

this chapter.

- S is the reference set for the state of the cells. This set is always non empty and countable, and usually of a limited cardinality. If we show the state of a cell $c_{\vec{i}}$ at time t is shown by $c_{\vec{i}}(t)$ then S will be as shown is Equation 3.2

$$S = \bigcup_{t \geq 0, c_{\vec{i}} \in C} \{c_{\vec{i}}(t)\} \tag{3.2}$$

  $S = \{0, 1\}$ forms a binary CA. The size of S is denoted as σ in this work. The state of each and every cell is updated simultaneously in a discrete time. CA is deterministic if the state of each cell at any discrete time t_i can be uniquely calculated starting from a certain initial configuration. In a probabilistic CA the probability of each cell $c_{\vec{i}}$ to be at the state S_j at the time t_k is a known value $p_{\vec{i}jk}$, such that $\forall \vec{i}, j, k : 0 \leq p_{\vec{i}jk} \leq 1$ and $forall\vec{i}, k : \sum_{j=1}^{\sigma} p_{\vec{i}jk} = 1$.

- N is the function that defines the cells' neighborhood as an ordered tuple of cells in C. For a neighborhood of size n, $C \overset{N}{\rightarrow} C^n$ and $N(c_{\vec{i}} \in C) = \left\langle c_{\vec{i}}^1, c_{\vec{i}}^2 \ldots, c_{\vec{i}}^n \right\rangle$ such that:

$$\forall c_{\vec{i}} \in C, \vec{\Delta} = [\delta_1, \delta_2, \ldots, \delta_D] : \; N(c_{\vec{i}}) = \left\langle c^1_{\vec{i}}, c^2_{\vec{i}} \ldots, c^{\eta}_{\vec{i}} \right\rangle \Longleftrightarrow \\ N(c_{\vec{i}+\vec{\Delta}}) = \left\langle c^1_{\vec{i}+\vec{\Delta}}, c^2_{\vec{i}+\vec{\Delta}} \ldots, c^n_{\vec{i}+\vec{\Delta}} \right\rangle \tag{3.3}$$

In other words if $N(c_{\vec{i}})$ is the neighborhood of $c_{\vec{i}}$ then $N(c_{\vec{j}})$, where $c_{\vec{j}}$ is a linear transform of the cell $c_{\vec{i}}$ in the cellular space C with the D-dimensional vector Δ, is an ordered tuple of the same length as of $N(c_{\vec{i}})$that includes the transformed of each and all members in $N(c_{\vec{i}})$ with the same vector Δ and in the same order. The n_{th} element in the neighborhood of a cell is called the n_{th} neighbor of the cell. Although not necessary, the neighbors of a cell are usually the cells with a short euclidean distance to that cell. It is common to define a *neighborhood radius* r [67],[33] such that:

$$\forall c_{\vec{i}}, c_{\vec{j}} \in C : \; \left| \vec{i} - \vec{j} \right| \leq r \Longleftrightarrow c_{\vec{i}} \in N(c_{\vec{j}}) \quad \wedge \quad c_{\vec{j}} \in N(c_{\vec{i}}) \tag{3.4}$$

η is the number of elements in the output ordered n-tuple of neighborhood function are fixed for all the cells and is called the *Neighborhood Size*. Each cell in the CA can read the state of all and only the cells in its neighborhood. If C is not circular on all its dimensions, it is possible that the neighborhood function returns one or more cell that are not located inside C. The state of such neighbors are decided in advance to be a fixed value in S. For example a binary CA with limited non-cyclic dimensions can assume that the state of any cell placed outside the boundaries of C is always zero.

The 'is a neighbor of' relation in its general form is neither symmetric nor anti-symmetric. However, relying solely on a neighborhood radius to define N makes it a symmetric relation. The neighborhood of a cell can include the cell itself. The *neighborhood configuration* of a cell at any specific time is a n-tuple in σ^{η} where the n_{th} element is the state of the n_{th} neighbor of the cell.

Example 3.1.

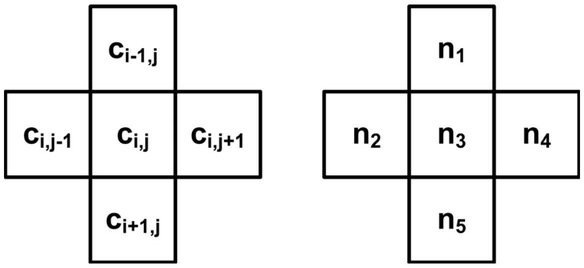


Figure 3.2: The neighborhood defined in Example 3.1

Assume a binary 2-D CA with the neighborhood $N(c_{[i,j]}) = \langle c_{[k,l]} \mid (i-k)^2 + (j-l)^2 \leq 1 \rangle$ *and*

$\forall c_{[k,l]}, c_{[m,n]} \in N(c_{[i,j]}) \ : \ c_{[k,l]} \prec c_{[m,n]} \Longleftrightarrow k < m \vee (k = m \wedge l < n)$

where $\prec$ *defines the order in the output 5-tuple* $\langle n_1, n_2, n_3, n_4, n_5 \rangle$ *of* N*. Figure 3.2 illustrates this neighborhood.*

- Finally, f is the transition function that inputs the configuration of neighborhood of a cell and outputs the state of the cell in the next time step: $S^\eta \xrightarrow{f} S,\ c_i(t+1) = f(N_i(t))$, where $c_i(t)$ is the state of the cell c_i at time t and $N_i(t)$ is the neighborhood configuration of c_i at time t. In case of a probabilistic CA, f assigns a probability p_i to each state s_i such that $\sum p_i = 1$. However, our focus will only be on deterministic CA. We can define the function $\mathcal{F} : \ C \xrightarrow{\mathcal{F}} C,\ C(t) = \mathcal{F}(C(t-1))$, where $C(t)$ is the configuration of the CA at time t where the state of each cell has been updated according to f.

The transition function is commonly expressed in a list of *If-Then* rules where the *if* part is a η-tuple of states in S and the *then* part is a single member of S. $\eta = |N|$ and each rule tells if the neighborhood configuration of a cell is the given η-tuple in the *if* part at time t, the state of the cell will be what is given in the *then* part at time $t+1$. If we denote the i_{th} element in S as s_i and $\sigma = |S|$, then f can be expressed as in Equation 3.5.

$$f = \begin{cases} \langle s_1 s_1 \ldots s_1 \rangle & \longrightarrow s_{out}^{1} \\ \langle s_1 s_1 \ldots s_2 \rangle & \longrightarrow s_{out}^{2} \\ \ldots \\ \langle s_1 s_1 \ldots s_\sigma \rangle & \longrightarrow s_{out}^{\sigma} \\ \ldots \\ \ldots \\ \langle s_\sigma s_\sigma \ldots s_\sigma \rangle & \longrightarrow s_{out}^{(\sigma^\eta)} \end{cases} \tag{3.5}$$

Given that all the possible configuration appear in the *if* part, one can omit them and store only s_{out}^{i}s with the specific order of Equation 3.5. Another acceptable form to store f for a binary CA (where $S = 0, 1$), is to store only the neighborhood configurations whose corresponding output is 1, usually in a decimal format (e.g. 19 instead of $\langle 010011 \rangle$). The former will take σ^η bits to be stored for a binary CA and the latter takes $\iota.\eta$, where ι is the number of neighborhood configurations for which s_{out} is 1.

Example 3.2.

Suppose a binary 2-D CA with the neighborhood defined in the Example 3.1 that sets the state of each cell to 1 if its neighborhood in the previous time step has been one of the 3 configurations depicted in the Figure 3.3. Given that a black square represents a cell in state 1 and a white square represents a cell in state 0, these neighborhood configurations can be translated to 25, 21 and 10 in order from left to right. This transition function can be written in either of the following forms:

1. $\sum 25, 21, 10$

2. $(00000010001000000000010000000000)_2 = 35652608$

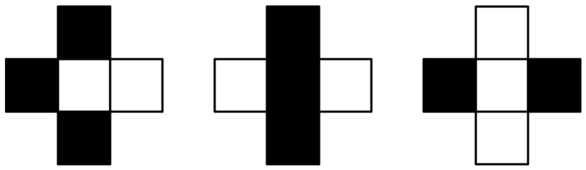


Figure 3.3: The neighborhood configurations that set the cell's next state to 1 (black) in Example 3.2

3.1.2 Cellular Automata With Memory

Cellular Automata introduced above is an memoryless or ahistoric. However, there has been studies that shows adding memory to the Cellular Automata might be helpful for specific tasks [3]. In short, a Cellular Automata with m-step memory can use the last m configurations $(C(t-m), C(t-m+1), \ldots, C(t))$ in its transition function to create the configuration of the CA at the next step $(C(t+1))$. If expressed as a rule set, the transition function will have m neighborhood configurations in each of its rules, each neighborhood configuration expressing a specific neighborhood configuration for one specific time step. The neighborhood of the cell (not to be mistaken with neighborhood configuration) does not change over time. Adding history to Cellular Automata makes more data available to the transition function and has a good chance to make the neighborhood smaller for an f expressed in form of Equation 3.5, because even a smaller neighborhood might have enough data to form f. In Section 3.3 we study the effects of adding memory to the Cellular Automatas for specific problems.

3.1.3 Applications of Cellular Automata: A More Detailed Review

We briefly reviewed some applications of the Cellular Automata in Chapter 1. Here we dive into more details necessary for a better understanding of the problem to be solved in this chapter.

3.1.3.1 Simulation Problems

Cellular Automata has been studied for simulating many complexed systems where global behaviors emerge from local simple interactions. One of the most prominent examples is the Conway's famous Game of Life [29, 2] whose patterns, rules and variations has been used for studying of dynamic systems [10, 50, 49, 67]. The transition function f in this class of applications is manufactured to model the local interactions in the system being studied. The initial configuration ($C(0)$) as well is manually designed to simulate the start point of the system. The CA then starts to go through its state space and in most cases human intelligence is involved to analyze the visualization or the statistical features of the CA over time. The other purpose of such experiment can be finding initial patterns with certain property such as self replication or interacting with other patterns to form a desired high-level behavior. The transition function nevertheless is pre-set in all problems of this class.

There exists another class of problems in the field of Cellular Automata where the final configuration of the CA (or set of the acceptable final configurations) is known and the transition function is searched for. This problem is usually referred to as the *inverse problem of Cellular Automata*. The initial state can be either preset or variable, according to the specific problem being solved. Two common cases of the inverse problem are classification and pattern generation in Cellular Automata as explained in Sections 3.1.3.2 and 3.1.3.3.

3.1.3.2 Classification Problems

The set of all ς^σ possible configurations of the CA$\langle C, S, N, f\rangle$ ($\varsigma = |C|, \sigma = |S|$) is forms the *state space* of the CA. For any CA:

$$\exists c, s, t_n \in \mathbb{N};\ |C| < c\ \wedge\ |S| < s\ \wedge\ t > t_n \Rightarrow \exists t_i, t_j : C(t_i) = C(t_j), t_i \neq t_j \tag{3.6}$$

Keeping in mind that $C(t) = \mathcal{F}(C(t-1))$, it can be concluded from Equation 3.6 that starting from any configuration in the state space, the CA with a finite C and S either settles down in a steady configuration or repeats a cycle of fixed configurations after a certain number of steps. These steady cycles are named *basins of attraction*[71] of CA. The transition function forms the basins of attraction in the state space. Let's define the relation $\prec$ as in the the Equation 3.7:

$$C(i) \prec C(j) \iff \exists n \in \mathbb{N} : C(j) = \mathcal{F}^n(C(i)) \tag{3.7}$$

Then $\prec$ partitions the state space into sets where all the elements have the same basin of attraction. This means that CA can solve classification problems as long as there is a mechanism to detect at least one configuration in the basin of attraction. the challenge will be to find a transition function f that forms the basins of attractions such that instances of two distinct class do not fall in the same basin of attraction, and ideally each class will have only one basin of attraction.

We remember from Section 3.1.1 that f is commonly represented as an ordered n-tuple of states where $n = \sigma^\eta$. The reference set of each element in this ordered n-tuple is S. For a known N and $C(0)$ (i.e. the initial state), the size of the search space for any single f will be σ^{σ^η}. In practice this number grows so quickly that finding the right transition function is feasible for only small η and σ. To make the problem even harder, the neighborhood function is not always known in advance and it is up to the algorithm to find the appropriate N. Examples of classification problems for which CA has been used are [54], [37], [12] and [18].

3.1.3.3 Pattern Generation Problems

In contrast to the computational applications of CA described in Section 3.1.3.1 and 3.1.3.2, pattern generation is an example of memory (in contrast to computation) applications for the CA. The dimension of the CA matches the dimension of the pattern to be generated, and the target configuration is a preset point in the state space. This point in the state space is not necessarily a basin of attraction. The initial configuration and the number of time steps for the CA to reach to the target configuration are either preset or is up to the algorithm to be set. Nevertheless, the initial configuration has often very low or no complexity (e.g. all the cells can be in the same initial state). One advantage of applying CA for pattern generation is data compression. The data size required to store a pattern is tied to the complexity of the pattern, and for certain patterns storing the pattern in the transition function of a Cellular Automata can save a considerable amount of memory. Cellular Automata has the ability of creating complex patterns from a simple initial configuration and an acceptable sized transition functions. This however is not the sole benefit of employing CA in pattern formation problems. Studies have shown that patterns that are described by providing simple initial configurations and transition functions demonstrate interesting features such as scalability and fault tolerant [14] .Similar to the discussion in Section 3.1.3.2 the search space for f can be extremely large for even moderate sized CAs. The inverse problem of pattern generation in Cellular Automata has not been studied thoroughly and the application of most of current methods is limited to one dimensional or very small 2-D CAs. We will suggest a new method for solving this type of problems; i.e. pattern generation on large 2-D Cellular Automata in this thesis. However, to verify the extents of the methods explained here we puch them to their limits by testing them on very complex patterns that have not been the result of a developmental method. This will help us to study and compare the ability of each method in more general cases. We talk more about the test cases in Section 3.1.4.

3.1.3.4 Challenges of the Inverse Problems

The main challenge of the inverse problem of the Cellular Automata is the very large search space for finding the appropriate transition function. Moreover, in order to reach a final configuration from an initial one, one should decide whether to get to the target configuration in fewer number of time steps, or with a more simple transition function. The answer might not always be the same for different problems. Even validation of a given transition function is not always a trivial task itself. We have to advance the CA in time for an exact number of times if we know the number of time steps. A considerable computation time is needed for this if the number of time steps, the size of cellular space and the neighborhood size are not small. Validation can be even harder when the number of required time steps to get to the final configuration is not known in advance. It is not always easy to tell if the CA's configuration is getting closer to the target configuration. A common distance measure for two configuration of a CA is the number of cells that are not in the same state in the two configuration. Equation 3.8 shows the distance function for a binary CA of size $m \times n$. However, it cannot be guaranteed that this measure makes the distance of the configurations of CA to the future configuration a descending function over time.

$$distance(C(t_1), C(t_2)) = \sum_{i=1}^{m}\sum_{j=1}^{n} \left|c_{[ij]}(t_1) - c_{[ij]}(t_2)\right| \tag{3.8}$$

Example 3.3.

Suppose the task of generating the 32×32 *binary pattern in Figure 3.4 from an all white pattern. During the search for the transition function* f *that does this in a CA such as* $CA_{ex_{3.3}} = \langle C, S, N, f\rangle$*, where:*

- $C = \left\{c_{[i,j]} \mid 0 \leq i, j < 32\right\}$

- $\forall c_{[i,j]} \in C:\ c_{[i,j]}(0) = 0$
- $\forall t : i \geq 32 \vee j \geq 32 \Rightarrow c_{[i,j]}(t) = 0$
- $S = \{0, 1\}\,(0 : white, 1 : black)$

- $N(c_{[i,j]}) = \left\{c_{[k,l]} \mid (i-k)^2 + (j-l)^2 \leq 2\right\}$

Given the transition function shown in Equation 3.9 enough time, it will generate the target pattern in 32 time steps. Although this transition function achieves the goal perfectly, as depicted in the Figure 3.5 the distance of the CA configuration to the final pattern is not a descending function. If the search algorithm assumes that a good transition function should make the CA configuration closer to the target in each time step, it will miss the the valid transition function of Equation 3.9 (N_0 *and* N_1 *are defined in the Equation 3.10).*

$$c_{\vec{i}}(t) = \begin{cases} 0;\ N_{\vec{i}}(t-1) \in N_0 \\ 1;\ N_{\vec{i}}(t-1) \in N_1 \\ c_{\vec{i}}(t-1);\ otherwise \end{cases} \tag{3.9}$$

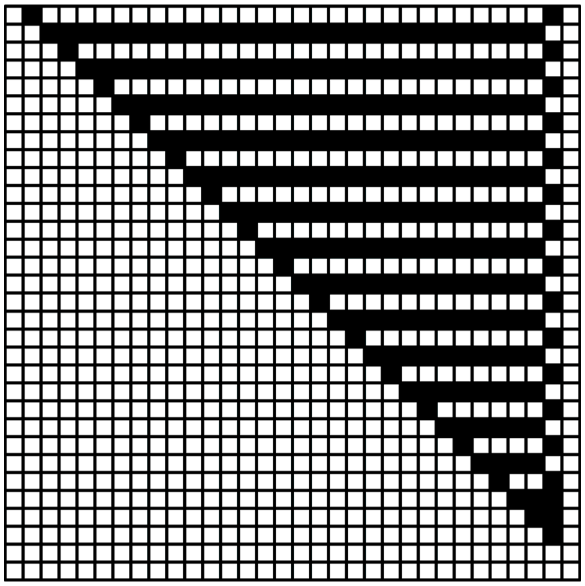


Figure 3.4: a 32×32 binary pattern generated from a blank (all white) pattern by the CA in Example 3.3

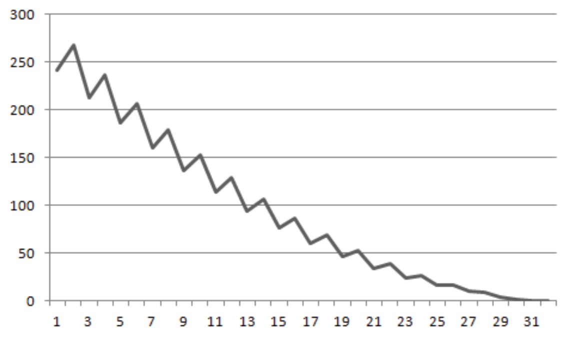


Figure 3.5: The distance between the configuration of the CA in Example 3.3 to the final pattern in the Figure 3.4 over time

$$\begin{aligned}
N_0 = &\quad \{\langle 1,1,1,1,1,1,0,0,0\rangle, \langle 0,1,0,1,1,0,0,1,0\rangle\} \\
N_1 = &\quad \{\langle 0,0,0,1,1,1,0,1,0\rangle, \langle 1,1,1,0,0,0,0,0,0\rangle, \\
&\quad \langle 1,1,0,0,0,0,0,0,0\rangle, \langle 1,1,1,0,1,0,0,0,1\rangle\}
\end{aligned} \tag{3.10}$$

3.1.4 Problem Statement

This thesis targets the pattern generation problem explained in Section 3.1.3.3. This problem is often studied for two consequent configurations of the CA because of the challenges

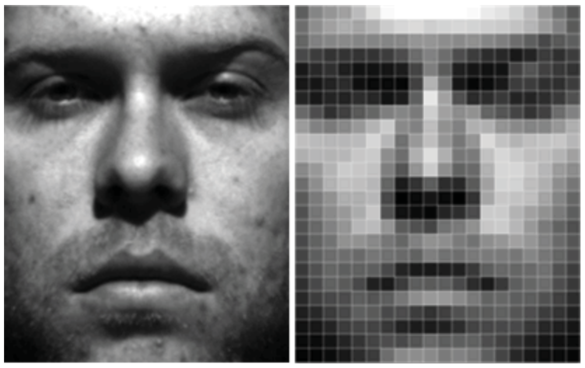


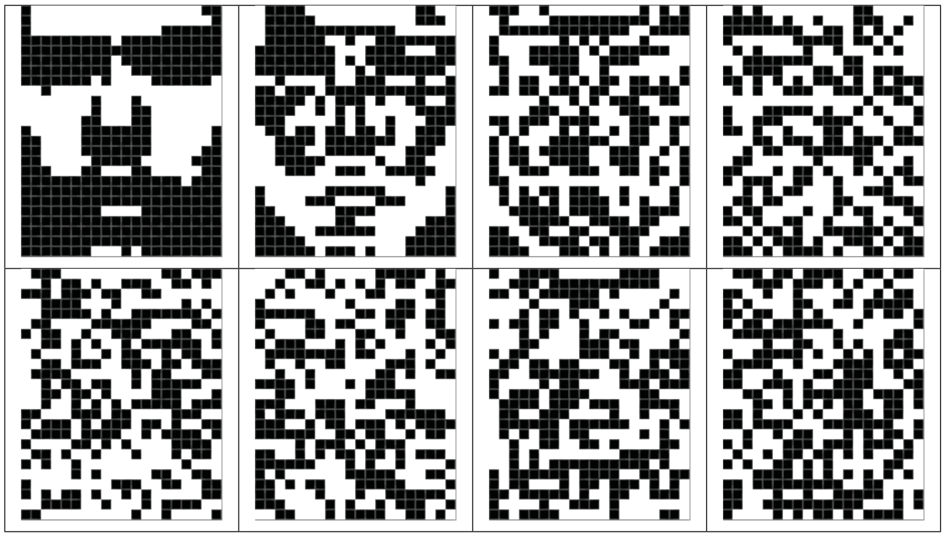
Figure 3.6: Resizing the image to reduce the computation time. The pixels of the right image are 100 times larger than the pixels of the left image.

mentioned in the Section 3.1.3.4. The problem statement is, in a CA$\langle C, S, N, f\rangle$ with given C, $C(t)$ and $C(t+1)$ and $S = \bigcup_{c_{\vec{i}}\in C}^{t\leq \acute{t}\leq t+1} c_{\vec{i}}(\acute{t})$, what are the f and N that produce $C(t+1)$ from $C(t)$. We review the current method, suggested improvements and our suggested methods in Section 3.2. We do not aim to reverse engineer and find a CA to generate an artificial pattern that already has been created by a well defined CA, but we prefer to have our method applicable to generate real life patterns that have not been resulting from applying any known CA. Such pattern will let us to measure and compare the performance of different methods without having a bias for any specific form of CA.

The pattern set to be generated here uses the Extended Yale Face Database B used in [19]. The images are resized to 20×25 pixels to keep the computation time acceptable, so we can repeat the experiments for a large number of images. 128 images are converted to 8-bit, 256-level gray scale images (Table A.1). Each image is divided to 8 layers, where each layer in presented as a binary black and white image. The pixel p is white in the coordinate (row_p, col_p) of layer l if the l^{th} bit in the 8-bit value of pixel located in the coordinate (row_p, col_p) of the reference 256-level gray scale image is one. Pixel p is black otherwise. Table 3.1 shows the resulted eight black and white image resulting from the image in Figure 3.6. Each black and white image is called a *layer* in this work.

Given the first layer (i.e. the layer visualizing the most significant bit, the upper left image in Table 3.1), the problem is to store the consequent layers in a format where the value of each pixel in any of the layers is determined according to the local information

Table 3.1: 8 black and white images created from the image in the Figure 3.6



available to the pixel in the previous layer (or the previous layers, in case of a CA with memory). Provided with the layer l, this format should be able to restore the layer $l+1$ with absolutely no error. The whole 256-level gray scale image can then be restored from the first black and white layer in the layer sequence.

This problem is translated to the CA domain by setting C to a 2-D, non cyclic cellular space with its size equal to the image size. The CA configuration at time t is equivalent to the pattern on $layer_t$, where $layer_0$ is the layer that visualizes the most significant bit of the 256-level gray scale image. S is the binary set $0, 1$ and we search for the N and f for each pair or consequent CA configurations (or for each combination of a configuration and its $h+1$ immediately previous configurations, in case of a CA with h steps memory).

The are several criteria for comparing different solutions, among which we can mention the required time to find the answer, required memory to find the answer and the required memory to save the answer. Keeping in mind that the answer needs to be found only once and off-line in most of the cases (i.e. not in real time), the required memory to store the solution seems to be the most important criteria for comparison. This is supported by the fact that the answer contains f and N, both of which need to be stored in every single cell in C. Any small change in the memory requirement of these two will be amplified by the number of cells who will be storing them. If it takes b_1 bits to store f when represented as

f_1 and b_2 bits when represented as f_2, There is a difference of $\varsigma \times |b_1 - b_2|$ bits to store only the transition function. The same argument applies to N, beside if the pattern-generator is about to get implemented on hardware, smaller the N is less the number of inter-cellular connections will be. when amplified by the number of cells in C, resulting in a major impact on the cost of hardware.

The above argument concludes that the memory requirement for storing the solution (f and N, to be more precise) is a good candidate to compare the outcome of different suggested methods here. Each suggested method is tried on different sets of inputs and different parameters are set for each set of input. New methods to solve this problem as well as improvements to existing methods are introduced in the Section 3.2 and their performances are compared in the Section 3.3.

3.2 Methodology

3.2.1 Expanding Neighborhood

The inverse problem of Cellular Automata is often studied for two consequent configurations of the CA because of the challenges mentioned in the Section 3.1.3.4. The problem statement is, in a CA$\langle C, S, N, f\rangle$ with given C, $C(t)$ and $C(t+1)$ and $S = \bigcup_{c_{\vec{i}} \in C}^{t \leq \acute{t} \leq t+1} c_{\vec{i}}(\acute{t})$, what are the f and N that produce $C(t+1)$ from $C(t)$. There has been studies such as [1] where N is known in advance and the focus is on optimizing the algorithm to find f. We remember from the Section 3.1.1 that the transition function is commonly represented as a set of *if-then* rules. To simplify the representation of the transition function depicted in the Equation 3.5, we show each rule as a $\left\langle \vec{NC}, s_{out} \right\rangle$, where $\vec{NC}$ is a specific neighborhood configuration representing the *if* part of the rule and s_{out} is a specific value from S, representing the *then* part of the rule. There are always one or more rule in the rule base and the neighborhood size is always equal or larger than zero. One of the detailed recent studies where both f and N are to be found is [62], where a framework for solving the inverse problem for both deterministic and probabilistic CAs is presented. Algorithm 3.1 expresses the core idea for deterministic CAs in simple terms.

Algorithm 3.1 Expanding Neighborhood Algorithm

```
1: Initiate the neighborhood to include no cell: $N = \phi$ or $\eta = 0$
2: For each cell, form a $\left\langle \vec{NC}, s_{out} \right\rangle$ pair such that the $\vec{NC}$ is the current neighborhood
   configuration in $C(t)$ and $s_{out}$ is the state of the cell in $C(t+1)$. Form $f$ to include
   all the created pairs: $f = \bigcup_{c_{\vec{i}} \in C} \left\langle N_{c_{\vec{i}}}(t), c_{\vec{i}}(t+1) \right\rangle$. The upper limit for the number of
   rules will be the number of cells in $C$.
3: If there is no conflict in the transition function formed in the Step 2, go to Step 4 (there
   is a conflict in $f$ if and only if $f$ includes at least two separate rules with identical $\vec{NC}$
   and different $s_{out}$s). Otherwise, extend the neighborhood to include one of the closest
   cells to the given cell: $N(c_{\vec{i}}) \Leftarrow N(c_{\vec{i}}) + \left\langle c_{\vec{j}} \right\rangle$ such that $\forall j, k, c_{\vec{j}}, c_{\vec{k}} \in C : \left|\vec{j} - \vec{i}\right| \leq \left|\vec{k} - \vec{i}\right|$
   and go to Step 2
4: Starting from the oldest element in $N$ to the most recently added element, eliminate all
   the unimportant elements in all $\vec{NC}$s. An element in $N$ is said to be important for a
   conflict-free transition function $f$ if discarding that element from $N$ introduces conflicts
   to $f$. An element $N_i \in N$ is older than $N_j \in N$ if $N_i$ was added to $N$ before $N_j$ in Step
   3.
```

3.2.1.1 Complexity analysis

For a CA with $|C| = \varsigma$ cells and $|N| = \eta$ neighborhood size, Algorithm 3.1 needs to verify f for a total of η times to be conflict free (after adding each neighbor). If S has $|S| = \sigma$ states, the number of *if-then* rules in the rule set of f (denoted by φ in the text) is $\min(\varsigma, \sigma^{\eta+1})$ in the worst case. Finding conflicts in a rule set of n rules has the time complexity of $O(n^2)$. Forming the rules in Step 2 for ς cells has the time complexity of $O(\varsigma)$. The first 3 steps of Algorithm 3.1 are then of time complexity $O(\varsigma) + O(\min(\varsigma^2, \sigma^{\eta+1}))$, or in other terms, $O(\eta\varsigma^2)$ if $\varsigma < \sigma^{\eta+1}$, or $O(\eta\sigma^{2(\eta+1)})$ otherwise. Note that η is growing during the algorithm but that does not effect our worst case analysis. In a realistic problem it is unlikely to have all the possible neighborhood configurations in $C(t)$. ς is usually smaller than $\sigma^{\eta+1}$ and a good estimation for the number of times two rules in f are checked for conflict in all the η iterations of the first part of the algorithm is $\eta \times \varsigma^2$.

In Step 4 of the Algorithm 3.1, η elements in the neighborhood are checked one by one for their importance, adding another $O(\eta)$ to the time complexity of the algorithm. Note that each step of the first part involves forming the whole rule base in f by reading $\varphi(\eta+1)$ cell states one by one from the CA in each step. A single step of the second part involves discarding an element from the already formed rule set. The two might take different times depending on the details of implementation, hence they are expressed separately.

3.2.1.2 Improvements to the Existing Method

Following methods are suggested to both speed of the algorithm and reduce the size of the transition function resulting from the Algorithm 3.1 by modifying steps 3 and 4.

Speeding Up the Algorithm: Algorithm 3.1 resolves the conflicts (i.e existence of two rules with the same $\vec{NC}$ and different s_{out}s) by extending N which results in adding new elements to the $\vec{NC}$. The algorithm blindly picks one of the closest cells and adds it to the neighborhood. Euclidean distance is used to measure the distance of the cells. However, it is quite possible that there are two or more cells with the same euclidean distance to the cell for which the neighborhood is being formed.

$$\begin{aligned}&\forall c_{\vec{i}}, c_{\vec{j}} \in C, \vec{i} = [i_1, i_2, ..., i_d], \vec{j} = [j_1, j_2, ..., j_d] :\\&\max_{1\leq k\leq d}(|i_k - j_k|) \leq r \Longleftrightarrow c_{\vec{i}} \in N(c_{\vec{j}}) \quad \wedge \quad c_{\vec{j}} \in N(c_{\vec{i}})\end{aligned} \tag{3.11}$$

Assume that in the step 3 of the algorithm, we have reached the neighborhood radius (defined in the Equation 3.11) r and there are still conflicting rules. The size of neighborhood is $(2r+1)^2$ at this point, and there are $2((r+1)+1)^2 - (2r+1)^2 = 8(r+1)$ cells with the distance $r+1$ from the center cell. Even if the conflict in the rule base can be resolved by adding one of them only, the algorithm needs to repeat step 3 on all of them in the worst case, as there is no guarantee that such conflict resolving cell is picked before the rest. The number of cells to be tried is $4(r+1)$ in the average case and 1 in the best case. In our modified version all of the closest cells are added to the neighborhood at once. This means checking for conflicts is done only r times for a neighborhood of radius r. This number is $(2r+1)^2$ in the worst case, $4(r+1)+(2r-1)^2$ on average case and $(2r-1)^2+1$ in the best case for the original algorithm. In all cases the complexity order of step 3 of the algorithm is decreased from $O(n^2)$ to $O(n)$. We could base our calculation on the number of neighbors (η) instead of r. That way the number of repetitions for step 3 is η for the original algorithm, and $\lfloor\sqrt{\eta}\rfloor + 1$ for the modified version.

The drawback of this modification is that there will be possibly more unimportant neighbors added to this neighborhood. However, the number of unimportant neighbors added to a neighborhood of radius r is $8(r+1)-1$ in the worst case, which is a linear factor for any neighborhood size. This drawback is neglectable specially for large neighborhoods.

Reducing the Size of the Results: In practice, it is not rare that $\varsigma < \sigma^{\eta+1}$ or in other terms, that not all the possible neighborhood configurations appear in the *if-part* of the rules at the beginning of Step 4 of Algorithm 3.1. Even if no neighborhood configuration appears twice in the entire binary 2-D CA with $\eta = 25$ (a 5×5 neighborhood) ς needs to be at least $2^{25} = 33554432$ to cover all the possible neighborhood configurations (on a square 2-D CA this means there are more than five thousand cells in each dimension). Since this is barely the case for the problems we are targeting in this thesis, we can safely assume that not all the possible configurations appear in the rule set of f. An immediate result of this is that the order of eliminating the unimportant neighbors in Step 4 in Algorithm 3.1 matters, as one or more initially unimportant neighbor(s) can turn into important neighbor(s). Example 3.4 provides a simplified transition function in which different order of discarding unimportant neighbors can lead to different results.

$$f : \begin{cases} \langle 000 \rangle \rightarrow 0 \\ \langle 011 \rangle \rightarrow 1 \\ \langle 101 \rangle \rightarrow 1 \\ \langle 111 \rangle \rightarrow 1 \end{cases} \tag{3.12}$$

Example 3.4.

Assume that the transition function is as depicted in the Equation 3.12 at the beginning of Step 4 of Algorithm 3.1 for a binary CA with $\eta = 3$. All of the three neighbors in this example are initially unimportant, since discarding any of them does not create any conflict. This is shown in the Equations 3.13, 3.14, 3.15.

$$\textit{discarding } 1^{st} \textit{ element}: \langle 00 \rangle \rightarrow 0,\ \langle 11 \rangle \rightarrow 1,\ \langle 01 \rangle \rightarrow 1 \tag{3.13}$$

$$\textit{discarding } 2^{nd} \textit{ element}: \langle 00\rangle \rightarrow 0,\ \langle 01\rangle \rightarrow 1,\ \langle 11\rangle \rightarrow 1 \tag{3.14}$$

$$\textit{discarding } 3^{rd} \textit{ element}: \langle 00\rangle \rightarrow 0,\ \langle 01\rangle \rightarrow 1,\ \langle 10\rangle \rightarrow 1,\ \langle 11\rangle \rightarrow 1 \tag{3.15}$$

However, eliminating the 3rd element makes the remaining two elements important since eliminating either of the remaining two elements will result in a conflict (Equation 3.16. The neighborhood size η is fixed at 2 at this step. Eliminating either of the 1st or the 2nd elements on the other hand will keep the other one still unimportant. The algorithm enters another iteration and only the 3rd neighbor will be remaining by the time the algorithm ends. Equation 3.17 shows the transition function after discarding the first element in the Equation 3.13.

$$\textit{discarding the } 1^{st} \textit{ element in Equation 3.15}: \langle 0\rangle \rightarrow 0,\ \langle 1\rangle \rightarrow 1,\ \langle 0\rangle \rightarrow 1 \tag{3.16}$$

$$\textit{discarding the } 1^{st} \textit{ element in Equation 3.13}: \langle 0\rangle \rightarrow 0,\ \langle 1\rangle \rightarrow 1 \tag{3.17}$$

Example 3.4 demonstrates that the order of eliminating the neighbors can result in different neighborhood size and therefore different transition functions. Decreasing the neighborhood size even by a very small factor can have a major impact on the efficiency of the solution because all the cells in C should store f. Keeping in mind that f is a rule base including up to ς number of *if-then* rules and the *if-part* of each rule includes η elements, and that the transition function is stored in every cell in CA, eliminating even one single neighbor from the neighbor will reduce the total CA memory requirement up to ς^2 in a memoryless CA, or $(h+1)\varsigma^2$ in a CA with h memory steps. Moreover, eliminating specific elements might make two initially different rules identical, saving even more in the amount of information to store the rule base. The first contribution of this section is to improve the Step 4 in the Algorithm 3.1 to result in a smaller neighborhood. In the most recent methods [62] the neighbors are examined linearly for being eliminated from the first added neighbor to the last added one. The reason is claimed to be that the last added element is always important (or it would not be added in the first place), and also that important neighbors are tend to be close to each other. While the first part is correct, the conclusion on the best order is not well-justified. Neither there has been any comparison between the results of different orders to eliminate the elements.

To reduce the size of the transition function and improve the memory requirement of the CA, we suggest two specific orders for examining the elements in the neighborhood in Step 4 of Algorithm 3.1. The first suggestion is to use the information gain of each element in the neighborhood to estimate its importance and start from the element with the least estimated importance. [36] describes the information gain of a property of a message instance as the amount of information that property contributes to the classification of that instance. According to Shannon's theory of communication [51], the total information in a set M of n messages with probability $P(m_i)$ for each message m_i is as defined in Equation 3.18.

$$I[M] = \left(\sum_{i=1}^{n} -P(m_i) \log_2 P(m_i) \right) = E[-\log_2 P(m_i)] \tag{3.18}$$

We can represent the transition function of the Equation 3.23 in the form of set of instances as depicted in Table 3.2. The instances there can be divided in two subsets M_0 and M_1 according to any of the neighbors, where the state of that neighbor is always 0 in M_0 and always 1 in M_1. In general, the expected information to tell the next state of the cell in all the created subsets can be calculated from the Equation 3.19 [36], $\eta = 4$ in this example, $|M_i|$ is the number of instances in the subset M_i and $I[M_i]$ is the total information of the subset M_i from the perspective of the next state of the cell. The gain from each neighbor n in a transition function which is represented as a set of instances M

is defined in Equation 3.20

$$E[n] = \sum_{i=1}^{\eta} \frac{|M_i|}{|M|} I[M_i] \tag{3.19}$$

$$gain(n) = I[M] - E[n] \tag{3.20}$$

Example 3.5. *Assume the transition function of Equation 3.23 that tells the next state of the cell to be either* 0 *or* 1*. According to Equation 3.18, the total information in the transition function is:* $-\frac{3}{5} \times \log_2 \frac{3}{5} - \frac{2}{5} \times \log_2 \frac{2}{5} = -0.6(-0.74) - 0.4(-1.32) = 0.97$ *bits.* $E[n]$ *is calculated in Equation 3.21 for each of the four neighbors. Having the* $E[n]$*, we can calculate the gain of each neighbor from Equation 3.20. Equation 3.22 shows the information gain of each neighbor. From there, we can sort the neighbors from the element with least information gain to the element with most information gain, resulting in the order* $< n_1, n_2, n_3, n_0 >$*. This will be the order according to which we will start examining the importance of the neighbors in step 4 of the Algorithm 3.1.*

$$
\begin{aligned}
E[n_0] =& \frac{3}{5} \times I[< 0101 >\to 1, < 0101 >\to 1, < 0101 >\to 1] \\
&+ \frac{2}{5} \times I[< 1110 >\to 0, < 1010 >\to 0] \\
=& 0.6 \times 0 + 0.4 \times 0 = 0 \\
E[n_1] =& \frac{2}{5} \times I[< 0011 >\to 1, < 1010 >\to 0] \\
&+ \frac{3}{5} \times I[< 0101 >\to 1, < 0110 >\to 1, < 1110 >\to 0] \\
=& 0.4 \times 1 + 0.6 \times (-\frac{2}{3} \times \log_2 \frac{2}{3} - \frac{1}{3} \times \log_2 \frac{1}{3}) \\
=& 0.4 + 0.6 \times 0.92 = 0.95 \\
E[n_2] =& \frac{1}{5} \times I[< 0101 >\to 1] \\
&+ \frac{4}{5} \times I[< 0011 >\to 1, < 0110 >\to 1, < 1110 >\to 0, < 1010 >\to 0] \\
=& 0.2 \times 0 + 0.6 \times 1 = 0.6 \\
E[n_3] =& \frac{3}{5} \times I[< 0110 >\to 1, < 1110 >\to 0, < 1010 >\to 0] \\
&+ \frac{2}{5} \times I[< 0101 >\to 1, < 0011 >\to 1] \\
=& 0.6 \times (-\frac{2}{3} \times \log_2 \frac{2}{3} - \frac{1}{3} \times \log_2 \frac{1}{3}) + 0.4 \times 0 \\
=& 0.6 \times 0.92 = 0.55
\end{aligned}
\tag{3.21}
$$

$$
\begin{aligned}
&gain(n_0) = 0.97 - 0 = 0.97 \\
&gain(n_1) = 0.97 - 0.95 = 0.02 \\
&gain(n_2) = 0.97 - 0.6 = 0.37 \\
&gain(n_3) = 0.97 - 0.55 = 0.42
\end{aligned}
\tag{3.22}
$$

$$f : \begin{cases} \langle 0101 \rangle \to 1 \\ \langle 0011 \rangle \to 1 \\ \langle 0110 \rangle \to 1 \\ \langle 1110 \rangle \to 0 \\ \langle 1010 \rangle \to 0 \end{cases} \tag{3.23}$$

Table 3.2: Representing the transition function in the Equation 3.23 as a set of instances. The parameters of each instance are the 4 neighbors n_0 to n_3 and the label.

Rule	n_0	n_1	n_2	n_3	Label
1	0	1	0	1	1
2	0	0	1	1	1
3	0	1	1	0	1
4	1	1	1	0	0
5	1	0	1	0	0

We suggest to use the information gain to sort the neighborhood elements for examining their importance because in a set of messages in form of binary sequences, those elements whose information gain is zero can be eliminated without any loss of data. We believe that the same concept might also hold true in a set of if-then rules of a binary logic function. If those instances include unimportant elements in the if part with zero information gain (i.e. constant in all the instances) we can eliminate them right away with no worries of the conflicts appearing in the rule set. An element with high information gain on the other hand has less chance to be unimportant so is less likely to be eliminated. Eliminating an element with high-information gain might force us to keep more than one element with low-information gain whose elimination might have been possible otherwise.

We will examine the neighbors for elimination according to their information gain (Equation 3.20) starting from the least informative neighbor to the most informative one. We need to keep in mind that eliminating one neighbor modifies the information gain of each remaining neighbor and therefore the information gains need to be updated after each elimination. This will be a computationally expansive process and will add a considerable time to the algorithm's run time. However, our experiments show that not re-sorting the neighbors dynamically during the Step 3 of the Algorithm 3.1 does not bring us any advantage for the size of resulted neighborhood. We will show this in details in Section 3.3.

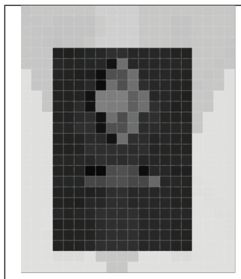
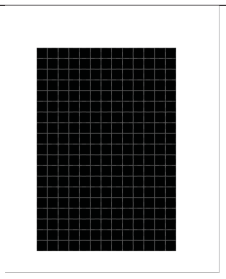
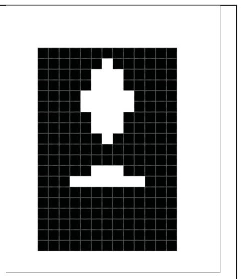
The results confirm our original assumption of the impact of information gain of a neighbor, as the resulted neighborhood is constantly and considerably smaller if we start examining for elimination from the least informative rather than the most informative neighbor. Example 3.6 demonstrates this for a simplified case.

The other idea to be examined is based on an important property of regular images, i.e. locality. In regular (not randomly generated images) cells (or pixels, in this context) located close to each other have similar characteristics. This makes us believe that if an element in the neighborhood is important, there should be a good chance that its adjacent element is important too. While the idea behind the previous method (sorting according to the information gain) was not specific to any type of pattern, the idea behind this suggestion is domain specific, i.e. is true for regular images. A second look at the step 4 of the Algorithm 3.1 shows that there is no guarantee that the physically close cells are added to the neighborhood at timely close steps. For example, two adjacent neighborhood elements with different distances from the center cell have a very small chance be added to the neighborhood at two consequent times.

To address the above issue, we examine sorting the cells according on their location in the CA. The neighborhood at the end of step 3 of the Algorithm 3.1 is formed as a square centered at the cell for whom we are finding the neighborhood. We start from one corner of this square and scan the neighbors either line by line or column by column. Note that we are building the neighborhood and the transition function in the step. Unlike running the CA, this step is done on a central processing module who has access to global CA properties such as the coordinate of the cells. Scanning the cells according to their location is a quite feasible task then.

The comprehensive results are presented and analyzed in the Section 3.3. It can be observed that elimination of the unimportant neighbors is considerably more efficient if they are examined for elimination this way, i.e. according to their location. Example 3.6

Table 3.3: An example image (left) and the patterns generated from its 1st bit (middle) and 2nd (right) bits

compares the effect of different orders explained here in the resulting neighborhood for a simple case.

Example 3.6.

Assume we are looking for the neighborhood and the transition function that generates the pattern of the second most significant bit of the right most image of the Table 3.3 from the pattern of its most significant bit. The patterns of the most and second most significant bits are depicted in the middle and the left images of the same table respectively. In step 3 of the Algorithm 3.1 the radius of the neighborhood is 9, i.e. there are $(2 \times 9 + 1)^2 = 361$ *elements in the neighborhood, most of them unimportant. We examine them one by one to eliminate the unimportant ones from the neighborhood. For a specific neighbor to be important or not depends on the order of examining the neighbors. We also remember from Example 3.4 that two or more initially different rules can merge into the same rule after one or more unimportant neighbors are eliminated. The number of rules in the transition function (represented as a rule set) will also depend on the order according to which the unimportant neighbors are eliminated. Table 3.4 compares the results of different orders for this matter. The complete results, comparison and analysis of the different orders is presented in the Section 3.3.*

One of the major issues of using the *Expanding Neighborhood* method is that the storage size of the resulted transition function is not scalable in general. By this, we mean if the size of the CA grows by a factor of n, the amount of memory required to store the transition function grows by a factor larger than n. In Section 3.2.2 and 3.2.3 we suggest two new formats for the transition function that not only let us to find the transition function much

Table 3.4: Comparison of the transition functions for the same CA using different orders in step 4 of the Algorithm 3.1.

Order of examining the neighbors	Number of final important neighbors	Number of rules in the transition function	Transition function size (bits)
Newest first	14	130	3640
Oldest first	16	63	2016
Least information-gain first	16	114	3648
Most information-gain first	19	110	4180
Linear, from the upper right corner down row by row	11	57	1254
Linear, from the lower left corner up row by row	12	65	1560
Random	17	150	5100

faster, but also contribute to the scalability, trying to keep the growth rate of the storage size of the transition function slower than the growth of the size of the CA.

3.2.2 First Suggested Method: Storing Individual Exceptions

3.2.2.1 Hidden Cell States and Hidden Transition Functions

It has been shown that finding the transition function might be easier if the cells have secondary states in addition to their main state [23] [21]. Such states provide extra information in addition to the neighborhood configuration to the transition function to determine the next state. We call such states *hidden states* here. When speaking of the CA configuration, it is only the collection of the main states of the cells that comes into account. The hidden states are invisible from the perspective of the CA configuration (hence the name *hidden*). In terms of the inverse problem of the Cellular Automata, the target pattern provides the cell's main state only. The problem is considered solve as long as the main states of the cells match the given pattern, no matter what the states of the hidden cells are.

From a more detailed perspective, the hidden states extend the state of the cell from a scalar value to a vector. The elements of the vector can be from different reference sets. However, instead of representing the overall state of the cell with a vector, we prefer to make a distinction between the main state (whose reference set is still S), and the vector of hidden states $\vec{h} = [h1, h2, \ldots, h\varphi]$ whose reference set is $S_{h1} \times S_{h2} \times \ldots \times S_{h\gamma}$ where γ is the number of hidden states in a CA. We show the value of the hidden states of the cell $c_{\vec{i}}$ at time t by $h1_{\vec{i}}(t), h2_{\vec{i}}(t), \ldots, h\gamma\vec{i}(t)$. Each hidden state is updated according to its own *hidden transition function*, which in terms takes input from all the hidden and main states of the cell itself and of its neighbors. Adding hidden states does not conflict the definition of CA presented in the Section 3.1.1, but merely provides a different representation to express complex transition functions in a more organized manner. The state sets of the hidden states have no obligation to be the same among the hidden or main states. The transition function still complies with the definition in the Section 3.1.1 as long as no hidden transition function accepts input from a non-neighbor cell or a neighbor cell in the current time.

Alian and Kharma [16] showed that adding $\gamma = D$ hidden states with the common reference set ($\vec{h}_{c_{\vec{i}}} \in \mathbb{N}^{\gamma}$) to a D-dimensional CA improves the evolvability of the CA. The hidden states are updated by simple local rules (i.e. transition functions). Although evolution was used there to find the transition function, we expect to benefit from such hidden states in our search as well. We try to find very simple hidden transition functions that produce enough information in the hidden states which can be used to resolve the conflicts we discussed earlier. Similar to [16], we set $S_{h1} = S_{h2} = \mathbb{N}$, and we try to resolve the conflicts by adding the minimum amount of information from the hidden states, which is the hidden states of one cell only.

We remember from the previous section that a conflict happens when two cells with the same state and the same neighborhood configuration at time t have different states at time $t + 1$. To resolve this conflict by the hidden state of the conflicting cells only, we need to guarantee that $\vec{h}$ is unique for each cell. We will use the hidden states *only* if there exists a conflict in the rule set of f. No hidden state will be used otherwise. While the unique combination of the hidden states can easily be used to explicitly determine the next main state of each cell, we stay cautious to use them as rarely as possible in the description of transition function. The reason is that relying solely on them will make the transition function huge, as the number of rules in f will be equal to the number of the cells in CA. Knowing that the hidden state values are integers and not binary numbers, and that the transition function should be stored in each and every cell of the CA supports this choice. Also, relying on the states of one cell only and discarding the neighborhood information defeats the main purpose of Cellular Automata and will most likely take away its major benefits such as scalability or fault tolerance.

The neighborhood size in the method explained here is always constant with a of radius 1 ($\eta = 9$) as defined in the Equation 3.4, so the inverse problem of CA (defined in the Section 3.1.3) is reduced to finding the transition functions, including the hidden and the main transition functions. The hidden transition functions are prefered to be as simple as possible to require minimum amount of memory in the cells. Also as said, they need to assign unique combination of states to each cell of the CA. Because there exist two hidden states $h1$ and $h2$ in a $2D$ CA, it will be enough if each hidden transition function generates unique values in one dimension only (Equations 3.24 and 3.25). The combination of the two hidden states will uniquely identify each cell.

$$\begin{aligned} &\forall [i,j],[i,k], c_{[i,j]}, c_{[i,k]} \in C \Longrightarrow h1_{[i,j]} = h1_{[i,k]} \\ &\forall [i,j],[k,j], c_{[i,j]}, c_{[k,j]} \in C \Longrightarrow h2_{[i,j]} = h1_{[k,j]}, \end{aligned} \tag{3.24}$$

$$h1_{[i,j]} = h1_{[k,j]} \Longrightarrow i = k \qquad h2_{[i,j]} = h1_{[i,k]} \Longrightarrow j = k \tag{3.25}$$

We can tell from the Equation 3.25 that the values of each hidden state should form a *cyclic group* [66]. There are limited number of bits in the cell to store the hidden state values. Because the overflow of the values stored in b bits with the *addition* operator is similar to an *addition modulo* 2^b, we can safely assume that the cyclic group is of addition form with modulo 2^b here. Any integer can be the group generator as long as it is relatively prime to 2^b. Example 3.7 shows how the cells in a 2D CA can be uniquely identified using two hidden states.

Example 3.7.

The cells in a 15×20 *CA can have two hidden states, one stored in 4 bits (*$2^4 = 16; 16 > 15$*)*

and one in 5 bits ($2^5 = 32; 32 > 20$). The values of the first hidden state can be generated as a cyclic group with the generator 13 ($gcd_{(13,16)} = 1$) and the values of the other can be generated as a cyclic group with the generator 9 ($gcd_{(9,32)} = 1$) with addition modulo 16 and addition modulo 32 respectively: $h1_{[1,j]} = 13 (j \in [1, 20]); h1_{[i,j]} = 13 + h1_{[i-1,j]} (i \in [2, 15])$ $h2_{[i,1]} = 9 (i \in [1, 15]); h2_{[i,j]} = 9 + h2_{[i,j-1]} (j \in [2, 20])$ *or in other terms:*

$h1 =< 13, 10, 7, 4, 1, 14, 11, 8, 5, 2, 15, 12, 9, 6, 3 >$

$h2 =< 9, 18, 27, 4, 13, 22, 31, 8, 17, 26, 3, 12, 21, 30, 7, 16, 25, 2, 11, 20 >$

The combination of $h1$ and $h2$ can uniquely identify any cell in the CA defined above.

We can extend the Example 3.7 to define the two hidden transition functions for a $d_1 \times d_2$ CA as in the Equation 3.26. Note that the hidden states $h1$ and $h2$ are stored in $\lceil \log_2 d_1 \rceil$ and $\lceil \log_2 d_2 \rceil$ bits respectively, so the normal addition will act similar to addition modulo $\lceil \log_2 d_1 \rceil$ and addition modulo $\lceil \log_2 d_2 \rceil$.

$$\begin{aligned} & h1_{[i,j]} = h2_{[i,j]} = 0 \ \ , c_{[i,j]} \notin C \\ & h1_{[i,j]} = h1_{[i-1,j]} + p_1 \ \ , gcd(2^{\lceil \log_2 d_1 \rceil}, p_1) = 1 \\ & h2_{[i,j]} = h2_{[i,j-1]} + p_2 \ \ , gcd(2^{\lceil \log_2 d_2 \rceil}, p_2) = 1 \end{aligned} \tag{3.26}$$

Notice the three important feature of the hidden transition functions defined in the Equation 3.26:

- They need access to only one cell in the neighborhood to create their output,
- They each create unique values in one dimension, so each cell in the CA can be uniquely tagged using the combination of both, and
- They do not change the value of the hidden states after the time step $\max(d_1, d_2)$ for any cell in the CA. in other terms, the hidden states stable after a finite number of time steps.

Unlike the main transition function that is expressed in form of a rule set with a variable number of rules and is different from a CA to another, the hidden functions can be constant for all CAs[16]. Any pair of functions that demonstrate the above listed characteristics are acceptable choices for any 2D CA independent of the CA configurations over time. Our method here adds a preparation phase to the solution of inverse problem, in which the hidden transition functions update the hidden states for at least $\max |d_i|$ number of steps, where d_is are the dimensions of CA and $|d_i|$ is the size of the CA in that dimension. There are different synchronization methods such as the Firing Squad[68] that guarantee all the cells start to update their main states once the hidden states are stable all over the CA. To avoid the details that do not directly concern our proposed method here, we can assume that the hidden states are stable after the preparation phase[16]. We call this new initial state C'_0. Replacing the initial CA configuration C_0 with C'_0, the main transition function can now use the information stored in the hidden states. We expect that this information prevents N to grow too large. It remains for the future researches to find the effect of hidden information on the required neighborhood size. Nevertheless, the neighborhood size is fixed to 9 and the number of hidden states is fixed to 2 for a 2D CA in this method.

Following list highlights the distinctive characteristics of our first proposed model:

- Cells have one main state and two hidden states, named $h1c_{[i,j]}$ and $h2_{c_{[i,j]}}$. We denote the states of these hidden states at time t by $h1_{[i,j]}(t)$ and $h2_{[i,j]}(t)$.
- The hidden state of the cell have integer values ($S_{row} = S_{column} = \mathbb{N}$)
- The hidden states get stabled to their fixed values in a preparation phase.
- N is always a neighborhood of radius 1 ($\eta = 9$), or $N(c_{[i,j]}) = \left\langle c^1_{[i,j]}, c^2_{[i,j]} \ldots, c^9_{[i,j]} \right\rangle$.
- The main states of each $c_{[i,j]} \in C$ is updated by the main transition function, whose output is always either 0 or 1 ($S = \{0, 1\}$).
- The main transition function does not take input from any hidden state of any neighbor cell in N other than the cell itself.
- The hidden transition function for each cell $c_{[i,j]}$ accept input from both main states and hidden states of all the cells in $N(c_{[i,j]})$.

- All the main and hidden states of all cells outside the boundaries of CA are fixed to a permanent pre-defined state.

3.2.2.2 Main Transition Function

Unlike the rules of the transition function in the existing methods, our type of rule has three elements and are represented by the ordered triple $\left\langle \vec{NC}, s_{out}, E \right\rangle$. $\vec{NC}$ is the main neighborhood configuration, a vector whose elements are the main states of the neighbor cells (e.g. $\langle s_1 s_2 \dots s_9 \rangle$). Also $s_{out} = c_{[i,j]}(t+1)$ is the main state of the cell at the next time step, and E is the set of exceptions to the rule that describes the combinations of the hidden states in the cell itself (and not in any of its neighbors). For a CA with H number of hidden states, E will be a set of H-dimensional vectors that specifically determine the combinations of the hidden states. For a cell with main neighborhood configuration $\vec{NC}$, the main state of the cell in the next time step (i.e. $c_{[i,j]}(t+1)$) will be $1 - s_{out}$ if the hidden states of $c_{[i,j]}$ at time t match any of the specific vectors mentioned in E. This can be expressed as in the Equation 3.27 for a 2D CA with two hidden states $h1$ and $h2$, where $s_i, s_{out} \in S = 0, 1$ and $h1_i$s and $h2_i$s are integer numbers.

$$\begin{aligned} \langle s_1 s_2 \dots s_9 \rangle \quad &\longrightarrow \quad s_{out} \\ unless: \quad &\vec{h} \in \{(h1_0, h2_0), (h1_1, h2_1), \dots, (h1_n, h2_n)\} \\ &(0 \leq n \leq \varsigma) \end{aligned} \tag{3.27}$$

Algorithm 3.2 explains the method to form the rule base for two consequent CA configurations at times t and $t+1$.

For each rule, we call the cells whose neighborhood configuration is $\vec{NC}$ and their hidden states values matches any of the vectors stored in the set E an *irregular cell* of that rule, in contrast to the rule's *regular* cells who have the same neighborhood configuration $\vec{NC}$ and their hidden state values do not mach any of the vectors in E.

Algorithm 3.2 Forming the Rule Base In the Storing Exceptions Method

1: For each cell in C such as $c_{[i,j]}$, form a triplet of the format $\left\langle \vec{H}, \vec{NC}, s_{out} \right\rangle$ where $\vec{H} = (h1_{[i,j]}(t), h2_{[i,j]}(t))$, $s_{out} = c_{[i,j]}(t+1)$ and $\vec{NC}$ is defined in the Equation 3.28

$$\begin{aligned}\vec{NC} = \langle & c_{[i,j]}(t), c_{[i-1,j]}(t), c_{[i-1,j+1]}(t),\\ & c_{[i,j+1]}(t), c_{[i+1,j+1]}(t), c_{[i+1,j]}(t),\\ & c_{[i+1,j-1]}(t), c_{[i,j-1]}(t), c_{[i-1,j-1]}(t)\rangle\end{aligned} \tag{3.28}$$

$h1_{[i,j]}(t)$ and $h2_{[i,j]}(t)$ are the hidden states of the cell $c_{[i,j]}$ at time t and $c_{[i,j]}(t)$ is the main state of $c_{[i,j]}$ at time t. There will be ς number of triplets at this point.

2: Divide the triplets to partitions where all the sectors in a partition have the same $\vec{NC}$ part. The number of partitions will be anything between 0 to ς.

3: Create one rule from each partition as follows and put the created rule in the rule set of transition function:

- If all the triplets in the partition have the same s_{out}, create a rule in form of $\left\langle \vec{NC}, s_{out}, \phi \right\rangle$ (ϕ is the null set).

- If both $\left\langle \vec{H}, \vec{NC}, 0 \right\rangle$ and $\left\langle \vec{H}, \vec{NC}, 1 \right\rangle$ co-exist in the triplets in the partition, divide the triplets in the partition to two sets H_0 and H_1 such that all the triplets in each set have the $s_{out} = 0$ and $s_{out} = 1$ respectively. Create the two sets H_0 and H_1 as $H_0 = \bigcup_{H \in \langle \vec{H}, \vec{NC}, 0 \rangle} H$ and $H_1 = \bigcup_{H \in \langle \vec{H}, \vec{NC}, 1 \rangle} H$. E will be the set with the lower cardinality (Equation 3.29).

$$E = \begin{cases} H_0; |H_0| \leq |H_1| \\ H_1; otherwise \end{cases} \tag{3.29}$$

Create a rule of the form $\left\langle \vec{NC}, s_{out_T}, E \right\rangle$, where s_{out_T} is assigned per Equation 3.30.

$$s_{out_T} = \begin{cases} 0; |H_0| \leq |H_1| \\ 1; otherwise \end{cases} \tag{3.30}$$

3.2.2.3 Complexity Analysis and Memory Requirement

This main characteristic of the method explained above is to add hidden states to the cells so they can be used later to create exceptions to the *if-then* rules. We remember from the Section 3.2.1 that for a transition function expressed in form of a rule set, a conflict rises when two individual rule assign different outputs to the same *if* parts, and in order to resolve the conflict we need to somehow differentiate between the two *if* parts. While the current existing methods try to do so by growing N for all the rules to the point that the two conflicting rules do not have the same *if* part anymore, this method differentiates between the conflicting rules by taking into account their hidden states. It ends up merging the two or more conflicting rules in one and registering the specific exceptions of that rule in the set E. The neighborhood size in the current existing methods (Algorithm 3.1) is increased for *all* the rules as long as there exists even only one pair of conflicting rules in the rule set. Keeping in mind that all the cells should store the whole rule set of the transition function, even small increases in the neighborhood size will result in considerable memory consumption. On the other hand, the price for keeping the neighborhood small is to store the exceptions, i.e. set of hidden state values represented as integer pairs of numbers. This is contrary to the neighborhood elements that are usually binary values and they require considerably more memory each. However, unlike the current existing methods that expand the neighborhood for *all* of the rules, the exceptions are stored *only* for the conflicting rules. Different rules can have different number of exception vectors in E. The comprehensive results and analysis of this method are presented in the Section 3.3. Meanwhile, Example 3.8 presents the results of using this method for the same problem of the Example 3.6.

The first step in Algorithm 3.2 is of complexity order $O(\varsigma)$. The partitioning in Step 2 has the complexity of order $O(\varsigma^2)$. The maximum number of partitions at the beginning of Step 3 is ς, making that step of complexity order of $O(\varsigma)$. The whole algorithm will then be of complexity order $O(\varsigma^2)$, which is clearly less than the complexity of the Algorithm 3.1, i.e. $O(\eta\varsigma^2)$ for its first 3 steps only. In addition to that, the Algorithm 3.1 has a final step to remove the unimportant neighbors. That step adds a considerable time to the running time of the algorithm, something that Algorithm 3.2 is free of. The complexity of the preparation phase explained in the Section 3.2.2.1 is of a linear order, which is negligible comparing to the complexity of the final step in the Algorithm 3.1. The actual measured run time for the two algorithms is presented in Section 3.3, where it can be seen that the run time of the Algorithm 3.2 has a meaningful advantage over the variations of the Algorithm 3.1.

The memory requirement of the Algorithm 3.2 depends on the specific CA configurations at time t and $t+1$. The memory required for one rule of the form $\left\langle \vec{NC}, s_{out}, E \right\rangle$ is $\eta \log_2 \sigma + log_2 \sigma + |E| \times (\log_2 \max_{c_{[i,j]} \in C} h1_{[i,j]} + \max_{c_{[i,j]} \in C} h2_{[i,j]})$, where $\eta = |\vec{NC} = 9$, $\sigma = |S| = 2$, $|E|$ is the number of vectors stored in the rule, and $\log_2 \max_{c_{[i,j]} \in C} h1[i,j]$ and $\max_{c_{[i,j]} \in C} h2[i,j]$ are the number of bits required to store the hidden states $h1$ and $h2$ respectively. Five bits will be enough to store the hidden state values for the test CAs mentioned in the Section 3.1.3.4. As the results in the Section 3.3 show, this method leads to noticeably smaller transition functions in terms of number of bits needed to be stored.

Another issue to be remembered when comparing this method with the Expanding Neighborhood method is that of inter-cellular communication in the CA. The Expanding Neighborhood method is superior to the Storing Exceptions method mainly in the cases where adding one or few *non-immediate* neighbors can resolve the conflicts in the transition function rule set. We emphasize on the non-immediate part because the Storing Exceptions method includes only the immediate neighbors. Although the Expanding Neighborhood method might appear to need less resources (i.e. no memory is spent on hidden states) we need to pay attention to the communication cost between the cells. Communication cost grows very quickly as soon as the distance between the two communicating cells grows on a real distributed platform. Having distant neighbors in the neighborhood means dedicating many more resources (e.g. transistors, bus, bus drivers, etc) for routing the signals, less switching speed and less clock frequency because of larger capacitors and resistance in longer paths, and more power to drive the circuit on an actual platform. On the other side, the Storing Exception method guarantees that neither the main state nor the hidden states need to communicate with any cell other than their immediate neighbors.

Example 3.8.

Assume the same problem of finding the transition function that develops the pattern of the right-most figure in table 3.3 from the middle figure in the same table. Using the Storing Exceptions *method explained here with* $p1 = p2 = 1$ *in the Equation 3.26 leads to the transition function demonstrated in the Equation 3.31. Table 3.5 compares this transition function with of the best transition function from the Table 3.4. The Storing Exceptions method in this example finds the transition function faster, and is also superior from the memory requirement perspective. Note that this method not only keeps the neighborhood size small, but also the number of rules in the transition function is considerably lower.*

$$f = \begin{cases} \langle 0,0,0,0,1,1,0,1,1\rangle \quad \longrightarrow \quad 1 \\ \\ \langle 0,0,0,1,1,1,1,1,1\rangle \quad \longrightarrow \quad 1 \\ \\ \langle 0,0,0,1,1,0,1,1,0\rangle \quad \longrightarrow \quad 1 \\ \\ \langle 0,1,1,0,1,1,0,1,1\rangle \quad \longrightarrow \quad 1 \\ \\ \langle 1,1,1,1,1,1,1,1,1\rangle \quad \longrightarrow \quad 1 \\ \mathit{unless} : \vec{h} \in \{\, (5,9),(6,8),(6,9),(6,10),(7,8),(7,9), \\ \qquad (7,10),(8,7),(8,8),(8,9),(8,10),(8,11), \\ \qquad (9,7),(9,8),(9,9),(9,10),(9,11),(10,8), \\ \qquad (10,9),(10,10),(11,8),(11,9),(11,10), \\ \qquad (12,9),(15,8),(15,9),(15,10),(16,6), \\ \qquad (16,7),(16,8),(16,9),(16,10),(16,11), \\ \qquad (16,12)\,\} \\ \\ \langle 1,1,0,1,1,0,1,1,0\rangle \quad \longrightarrow \quad 1 \\ \\ \langle 0,1,1,0,1,1,0,0,0\rangle \quad \longrightarrow \quad 1 \\ \\ \langle 1,1,1,1,1,1,0,0,0\rangle \quad \longrightarrow \quad 1 \\ \\ \langle 1,1,0,1,1,0,0,0,0\rangle \quad \longrightarrow \quad 1 \end{cases} \tag{3.31}$$

Table 3.5: Comparing the best results of the existing methods (Expanding Neighborhood) and the results of the Storing Exceptions method.

Method	Number of final important neighbors	Number of rules in the transition function	Transition function size (bits)
Best case of the *Expanding Neighborhood* from the Table 3.4	11	57	1254
Storing Exceptions	9	9	520

3.2.3 Second Suggested Method: Using Range of Values to Store Exceptions

We previously mentioned that the existing method of Expanding Neighborhood has the drawback of expanding the neighborhood for all rules to resolve a conflict that occurs for two rules only. Our first suggested method - Storing Exceptions - addresses this issue by storing the hidden state values of the cells that cause conflicts in the rule set. For some CA configurations however the number of stored exceptions in the part E of the rule might be too large. This consequently will make the whole rule set of f too large to be practical. In this section we propose a new method to keep both the neighborhood (hence $\vec{NC}$) and E small. For this, we keep the rule format similar to the rule format in the previous method, i.e. $\left\langle \vec{NC}, s_{out}, E' \right\rangle$. The first two elements are the same as described in the Equation 3.27. We look for a method to form E' from E, such that it represents the same irregular cells but in an alternative representation that require less memory to be stored. The rest of this section will be the search for such method. Once found, that method will form an extra step at the end of the Algorithm 3.2 to convert E to E' only.

We saw that E in a rule of the previous method stores the set of individual vectors of hidden state values for which the output of the owning cell will be the complementary to s_{out}. Each element e_i of the set stored in E can be assumed as a condition that reads: *if $\vec{h}$ is equal to e_i then the cell is an exception.* A promising idea will be to group together the conditions on the individual vectors of hidden states values, and to form conditions that apply to multiple vectors of hidden states. The new form of conditions can be read: *if $\vec{h}$ has the condition e_i then it is an irregular cell.* For this, E' should be a set of conditions on the hidden states that meet the following requirements:

- Each condition should apply to more than one vector of hidden state values on average.
- The conditions should apply to the hidden state values of rule's irregular cells only, and not the regular cells (i.e. cells whose next main state follows the s_{out} stated in the rule).
- The memory required to store the new conditions should be less than the memory required to store the individual vectors of hidden state values.

A closer look at the results of the previous method - such as the transition function depicted in the Equation 3.31 - shows that it is likely for the irregular cells in a rule there to be physically close to each other. This also matches the observation of biological development of multicellular organisms [20], where physically close cells gain similar properties. Therefor it seems that we can achieve the target mentioned earlier (to form conditions that apply to multiple vectors of hidden states) if we can force the hidden transition functions to assign unique but *close* values to physically close cells. That way we can have the conditions that tell the *range* of each hidden state value, such that if each hidden state value is in the range mentioned in the condition, the cell will be an irregular cell of that rule. Of course the limits of each range should be selected carefully so the above-mentioned requirements to hold true. Our first goal then needs to be to find such hidden transition functions.

For a D-dimensional CA with $\vec{H} = [h_{d1}, h_{d2}, \ldots, h_{dD}]$, where the values of the hidden state h_{di} on the i$^{\text{th}}$ dimension are between h^1_{di} to $h^{MAX_{di}}_{di}$, suppose the three different cells $c_l = c_{[d_1^{cte1}, d_2^{cte2}, \ldots, d_i^l, \ldots d_D^{cteD}]}$, $c_m = c_{[d_1^{cte1}, d_2^{cte2}, \ldots, d_i^m, \ldots d_D^{cteD}]}$ and $c_n = c_{[d_1^{cte1}, d_2^{cte2}, \ldots, d_i^n, \ldots d_D^{cteD}]}$ (d_i^{ctei} s are arbitrary constant numbers in the acceptable range of d_i, the i$^{\text{th}}$ dimension) have the values of their i$^{\text{th}}$ hidden states equal to h^l_{di}, h^m_{di} and h^n_{di} respectively. We can tell the physically close cells have close hidden state values if for each dimension d_i the above values follow the condition in the Equation 3.32.

$$|c_l - c_m| < |c_l - c_n| \iff \left|h^l_{di} - h^m_{di}\right| < \left|h^l_{di} - h^n_{di}\right| \tag{3.32}$$

where $|c_l - c_m|$ is the Euclidean distance between the two cells c_l and c_m. We can conclude from the Equation 3.32 that any function that creates an order on the whole sequence of the hidden states of the cells on a single dimension with the regular algebraic *smaller than* relation can be the desired hidden transition function. This function of course should still follow the Equation 3.24 to be an acceptable hidden transition function.

The hidden transition function in the Equation 3.24 creates an order on the sequence of cells with the regular algebraic $<$ relation if and only if no overflow happens for any of the cells in the corresponding dimension. There are two ways to guarantee this, either to enlarge n in the modulo n addition or to reduce the generator of the cyclic group so that the hidden state values of all the cells get stable before any overflow happens. The former will result in using more bits to store the hidden state values, because the modular addition is simulated by the overflow of integers stored in limited number of bits. We have to pick the latter method and set the generator of the cyclic group to $p_i = 1$ as the only choice to keep the number of required bits as low as $\lceil \log_2 d_i^{MAX} \rceil$ on the ith dimension with the size d_i^{MAX}.

Having such hidden transition functions, we can form *irregular regions* by storing the borders of such regions only. For a d-dimensional CA, in each rule $\left\langle \vec{NC}, s_{out}, E \right\rangle$, E will be a set of irregular regions, each of which in form of a d-dimensional section of the CA where the cells with the neighborhood configuration $\vec{NC}$ will have their next main state set to $1 - s_{out}$ instead of s_{out}. Note that it is not necessary for all the cells in that region to have the neighborhood configuration $\vec{NC}$. They can have any neighborhood configuration as long as there is no cell with the neighborhood configuration $\vec{NC}$ whose next main state is s_{out}.

Any limited continues linear section of the space of a D-dimensional CA can be identified by $2 \times D$ edges in the D-dimensional space and therefore requires only $2 \times D$ number of D-dimensional points to be stored. To make the memory requirement even less, we consider the D-dimensional sections in form of hyper-cubes only, where each edge is parallel to one dimension of the CA. In other terms, we consider the sections that can be described by the range of their dimensions, such as $h_i^1 < h_i < h_i^2$, where h_i^1 and h_i^2 are the limits of that section on the dimension d_i. Such section of the D-dimensional space can be identified by D points instead of $2 \times D$ points. For the case of a $2D$ CA, each section will be either a horizontal or a vertical rectangle identified by its upper-left and lower-right corners. The format of a rule resulting from this method will be as depicted in the Equation 3.33.

$$
\begin{aligned}
&\langle s_1 s_2 \dots s_9 \rangle \quad \longrightarrow \quad s_{out} \\
&unless: \\
&\left(h_1^{begin_0} < h_1 < h_1^{end_0} \wedge h_2^{begin_0} < h_2 < h_2^{end_0} \dots \wedge h_D^{begin_0} < h_D < h_D^{end_0} \right) \\
\vee &\left(h_1^{begin_1} < h_1 < h_1^{end_1} \wedge h_2^{begin_1} < h_2 < h_2^{end_1} \dots \wedge h_D^{begin_1} < h_D < h_D^{end_1} \right) \\
&\dots \\
\vee &\left(h_1^{begin_n} < h_1 < h_1^{end_n} \wedge h_2^{begin_n} < h_2 < h_2^{end_n} \dots \wedge h_D^{begin_n} < h_D < h_D^{end_n} \right)
\end{aligned}
\tag{3.33}
$$

Where $h_i^{begin_j}$ and $h_i^{end_k}$ are integer numbers and $0 \leq h_i^{begin_j} \leq h_i^{end_j} \leq d_i^{MAX}$. E' will be the disjunction of all conjunctive forms in the Equation 3.33, although in the implementation only the $h_i^{begin_j}$ and $h_i^{end_j}$ parameters are stored. Each conjunctive form in E' defines an *irregular region*. The union of the irregular regions in a rule should cover all the rule's irregular cells, and the rule's irregular cells only. The number of irregular regions (n in the Equation 3.33) for a specific rule depends on the selected $h_i^{begin_j}$ and $h_i^{end_j}$ values. Example 3.9 demonstrates this matter for a 2D CA. The main problem to be solved for this method is to find $h_i^{begin_j}$ and $h_i^{end_j}$ values for each individual rule such that minimize n.

Example 3.9.

The transition function of the CA explained in the Example 3.8 is listed in the Equation 3.31. Of the nine rules in the rule set, E in a null set in eight. The corresponding E' will be empty for all of those eight rules; i.e. rules 1, 2, 3, 4, 6, 7, 8 and 9. E in the 5th rule however contains 34 elements. That rule can be converted to any of the forms presented in the Equation 3.34 and 3.35. As it can be observed from those equations, the number of conjunctive forms in E' can vary for the same rule (8 and 7 in those two equations respectively).

$$\begin{aligned}
&\langle 1,1,1,1,1,1,1,1,1\rangle \quad \longrightarrow \quad 1\\
&\mathit{unless}:\\
&(5 \le h_1 \le 5 \wedge 7 \le h_2 \le 7) \vee (6 \le h_1 \le 11 \wedge 8 \le h_2 \le 10)\\
&\vee (8 \le h_1 \le 9 \wedge 7 \le h_2 \le 7) \vee (8 \le h_1 \le 9 \wedge 11 \le h_2 \le 11)\\
&\vee (12 \le h_1 \le 12 \wedge 9 \le h_2 \le 9) \vee (15 \le h_1 \le 16 \wedge 8 \le h_2 \le 10)\\
&\vee (16 \le h_1 \le 16 \wedge 6 \le h_2 \le 7) \vee (16 \le h_1 \le 16 \wedge 11 \le h_2 \le 12)
\end{aligned} \tag{3.34}$$

$$\begin{aligned}
&\langle 1,1,1,1,1,1,1,1,1\rangle \quad \longrightarrow \quad 1\\
&\mathit{unless}:\\
&(5 \le h_1 \le 5 \wedge 7 \le h_2 \le 7) \vee (6 \le h_1 \le 11 \wedge 8 \le h_2 \le 10)\\
&\vee (8 \le h_1 \le 9 \wedge 7 \le h_2 \le 7) \vee (8 \le h_1 \le 9 \wedge 11 \le h_2 \le 11)\\
&\vee (12 \le h_1 \le 12 \wedge 9 \le h_2 \le 9) \vee (15 \le h_1 \le 15 \wedge 8 \le h_2 \le 10)\\
&\vee (16 \le h_1 \le 16 \wedge 6 \le h_2 \le 12)
\end{aligned} \tag{3.35}$$

Lemma 3.1. *The upper limit of the minimum amount of memory required to store E' generated from E of a rule of the form in the Equation 3.27 in any CA of any dimension is twice the amount of memory required to store E.*

Proof. In any rule of the form as of the Equation 3.27, each vector stored in the set of irregular cells in E can convert to a conjunctive form as of the Equation 3.33, forming a one-to-one mapping between the irregular cells and the irregular regions (each irregular region contains one cell only in this case). We remember from the section 3.2.2 that there are D elements in each vector of the set in E so a total of D elements need to be stored there. On the other side, each conjunctive form needs to store one upper limit and one lower limit for each of the hidden states, where there are D hidden states in a D-dimensional CA. The total number of elements will be $2 \times D$, twice the number of elements in the former case. □

As said earlier, the main focus of this section will be to come up with feasible algorithms that find a minimum set of irregular regions that satisfy the above said conditions; i.e. cover all the irregular cells, and the irregular cells only. We name this problem the *Minimum Irregular Regions Problem* and provide the formal definition for the case of a $2D$ CA in the Definition 3.1 to have a better understanding of it.

Definition 3.1. *Assume a continues finite discrete 2D space P with the dimensions X and Y and finite arbitrary sizes x_{MAX} and y_{MAX} on those dimension respectively (Equation 3.36).*

$$P \subset \mathbb{Z}^2, \quad (x, y) \in P \iff (0 \leq x \leq x_{MAX} \wedge 0 \leq y \leq y_{MAX}) \tag{3.36}$$

An object is defined in P with the tuple $\langle Label, Coordinate \rangle$ as explained in the Equation 3.37.

$$\begin{aligned}
&\forall \ \ o \left\langle Label_o, Coordinate_o \right\rangle : \\
&Label_o \in \{i, r\}, \\
&Coordinate_o = (x_o, y_o), (x_o, y_o) \in P
\end{aligned} \tag{3.37}$$

Two objects are equal if and only if they have the same $Label$ and the same $Coordinate$. Also for any two objects o_1 and o_2:

$$\begin{aligned}
&\forall \ \ o_1 \left\langle Label_1, Coordinate_1 \right\rangle, o_2 \left\langle Label_2, Coordinate_2 \right\rangle : \\
&Coordinate_1 = Coordinate_2 \iff o_1 = o_2
\end{aligned} \tag{3.38}$$

In other terms, the coordinate of the objects are unique; i.e. two or more objects cannot share the same coordinate.

A rectangle c is defined in $\mathbb{Z}^2$ with a pair of coordinates $(x^{ul}, y^{ul}), (x^{lr}, y^{lr})$ such that $x_c^{ul} \leq x_c^{lr}$ and $y_c^{lr} \leq y_c^{ul}$. A rectangle $c \left\langle (x_c^{ul}, y_c^{ul}), (x_c^{lr}, y_c^{lr}) \right\rangle$ is in the space P (denoted as

$P \boxplus c$) if and only if $0 \leq x_c^{ul} \leq x_c^{lr} \leq x_{MAX}$ and $0 \leq y_c^{lr} \leq y_c^{ul} \leq y_{MAX}$. A rectangle covers the object $o(x_o, y_o)$ in P (denoted as $c \boxdot o$) if and only if $x_{ul} \leq x_o \leq x_{lr}$ and $y_{lr} \leq y_o \leq y_{ul}$. Given the finite set I of objects whose Label $= i$ and the finite set R of the objects whose Label $= r$ where $0 \leq |R \cup I| \leq x_{MAX} \times y_{MAX}$, the Minimum Irregular Regions Problem *is to find the smallest set C of rectangles in P that cover all the objects in I and no object in R.*

The verification of a solution T_i to the above problem has the complexity order of $O\left(|R \cup I| \times |T_i|\right)$; i.e. it can be verified in a polynomial time. This problem is very similar to the *red-blue set cover problem* [11], which itself is a special case of the *minimum set cover problem* [28], a well-known NP-Complete problem. The red-blue set cover problem is stated as following[11]:

Definition 3.2. *Given a finite set of* red *elements R, a finite set of* blue *elements B and a family $U \subseteq \mathcal{P}(R \cup B)$ (where $\mathcal{P}(S) = 2^S$, denoting the power set of set S), the red-blue set cover problem is to find a subfamily $V \subseteq U$ which covers all blue elements, but which covers the minimum possible number of red elements.*

Lemma 3.2. *The Minimum Irregular Regions Problem is a special case of the red-blue cover set problem.*

Proof. It is obvious that the sets I and R in the minimum irregular regions problem are equivalent to the sets B and R of the red-blue cover set problem. Moreover, we can form the set C in the Minimum Irregular Regions Problem as in Equation 3.39.

$$\begin{aligned} C = \{c_i \left\langle (x_{c_i}^{ul}, y_{c_i}^{ul}), (x_{c_i}^{lr}, y_{c_i}^{lr}) \right\rangle \mid\ & 0 \leq x_{c_i}^{ul} \leq x_{c_i}^{lr} \leq x_{MAX}, \\ & 0 \leq y_{c_i}^{lr} \leq y_{c_i}^{ul} \leq y_{MAX}, \\ & \exists o \in P:\ c_i \boxdot o\ \} \end{aligned} \tag{3.39}$$

Let Γ be a mapping defined in the Equation 3.40. The result of such mapping on the set C produces a set $U \subseteq \mathcal{P}(I \cup R)$, such that the three sets U, I, R will be equivalent to the three sets U, B, R in the red-blue cover set problem. There is a one-to-one mapping between the elements in the C and U, so any solution to the equivalent red-blue set cover problem can be mapped to a subset of rectangles in C to produce the solution to the problem in Definition 3.1 using Γ^{-1}.

$$\Gamma : \bigcup_{P \boxplus c_i} c_i \longmapsto \mathcal{P}\left(R \cup I\right); \;\; \Gamma(c_i) = \{o_j | c_i \boxdot o_j\} \tag{3.40}$$

$$\begin{gathered}\Gamma^{-1}\left(U = \bigcup_{i=1}^{n} o_i(x_{o_i}, y_{o_i})\right) = c\left\langle (x_c^{ul}, y_c^{ul}), (x_c^{lr}, y_c^{lr})\right\rangle \Longleftrightarrow \\ x_c^{ul} = \min_{i=1}^{n} x_{o_i}, \;\; x_c^{lr} = \max_{i=1}^{n} x_{o_i}, \;\; y_c^{ul} = \max_{i=1}^{n} y_{o_i}, \;\; y_c^{lr} = \min_{i=1}^{n} y_{o_i}\end{gathered} \tag{3.41}$$

Although $\Gamma^{-1}\left(\Gamma(c_i)\right) \neq c_i$, the mapping and its inverse are enough to convert the Minimum Irregular Regions Problem to a red-blue set cover problem and convert the solution of the latter back to the former domain. However, Lemma 3.1 implies a condition on the set U in the equivalent red-blue set cover problem: there always exist a solution C such that it contains no red object. The reason is the way the set C is formed in the Equation 3.39, that guarantees for each object with $Label = I$ (equivalent to a blue object) there exist a rectangle that includes that sole object only. This implies that in the worst case the solution to the red-blue set cover problem will be the union of the Γ transformed of such rectangles; i.e. a union of the sets from $\mathcal{P}(B \cup R)$ that contain no red object. □

It is shown in [17] that the red-blue set cover problem is an NP-hard problem. However, we know that for any arbitrary input to the Minimum Irregular Regions Problem has only one rectangle in the best case (a rectangle that covers all the i objects and no r object), and $|I|$ rectangles in the worse case, where each rectangle covers only on i object. Moreover, we can form the set C in the Equation 3.39 so that it includes only the rectangles that do not cover any r object, and then convert the Minimum Irregular Regions Problem to a well-studied set cover problem and use a greedy algorithm to find the solution. Nevertheless, forming C in that way will have a complexity of non linear order itself $\left(O\left((x_{MAX}.y_{MAX})^2|I \cup R|\right)\right)$. In Sections 3.2.3.2 to 3.2.3.6we suggest two greedy and one evolutionary algorithm tailor designed for the Minimum Irregular Regions Problem.

3.2.3.1 Exhaustive Search of Minimum Irregular Regions

As a reminder from the Definition 3.1, we are searching for the smallest set of rectangles that cover all and only the irregular cells. Algorithm 3.3 present the exhaustive search for this problem in high-level terms:

Algorithm 3.3 Exhaustive Search For the Minimum Irregular Regions Problem

```
1: Form all the possible combinations of rectangles that cover all the irregular cells.
2: Among the above combinations select only those who do not cover any regular cell.
3: Among the above selection select the combination(s) with the minimum number of
   rectangles.
```

The number of total possible combinations in the first step grows so fast with the size of I that makes it impractical for even relatively small sets of I. To have a better understanding of the size of the search space we can impose a restriction on the combinations in the first step and consider only those combinations where each irregular cell is covered by one and only one rectangle. The number of combinations for the set I with $|I|$ elements in this case will form the Bell numbers [57] who can be written as in the Equation 3.42, where ϖ_n is the number of possible partitionings of a set with n elements. Note that $\varpi_0 = \varpi_1 = 1$.

$$\varpi_{|I|} = \sum_{k=0}^{|I|} \binom{|I|}{k} \varpi_{|I|-1} \tag{3.42}$$

"There is no known simple closed-form expression for ϖ_n" [57] but there are some asymptotic formulae that try to approximate the Bell numbers. The sequence *A000110* in the On-Line Encyclopedia of Integer Sequences (OEIS) [53] is of the Bell numbers and has the first 27 elements of the series as 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, 27644437, 190899322, 1382958545, 10480142147, 82864869804, 682076806159, 5832742205057, 51724158235372, 474869816156751, 4506715738447323, 44152005855084346, 445958869294805289, 4638590332229999353, 49631246523618756274. Figure 3.7 provides a logarithmic graph of the number of partitionings per number of elements in $|I|$. Even if the two next steps take small constant times, an exhaustive search

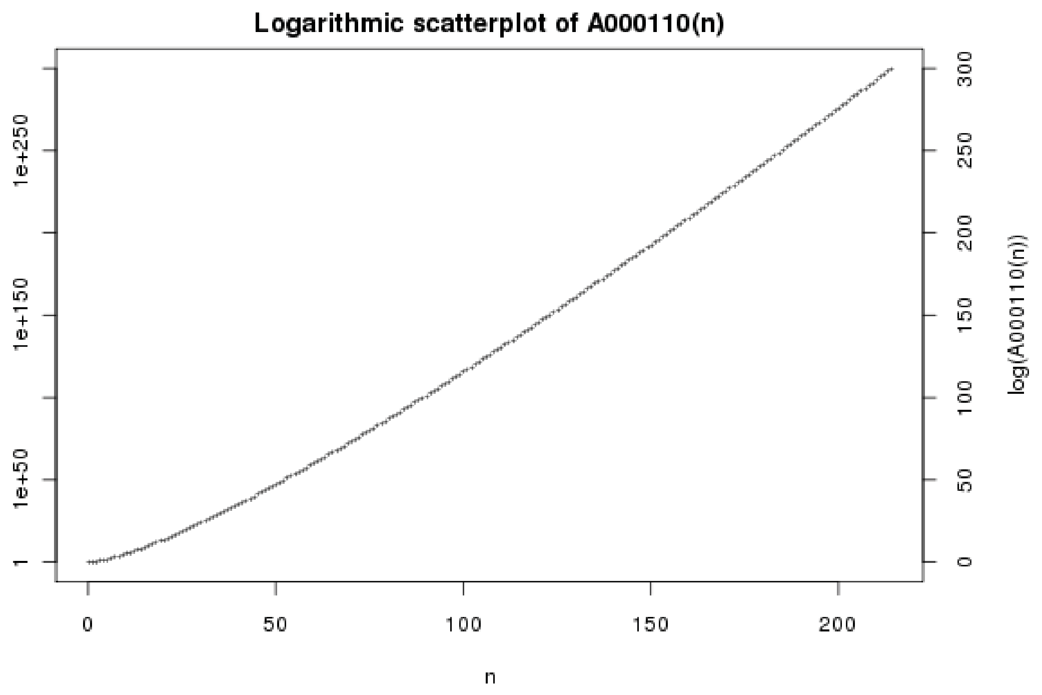


Figure 3.7: The growth of Bell numbers. Source: OEIS [53]

will be practically impossible.

We present three different algorithms in the rest of this section to solve the Minimum Irregular Regions Problem. The first two are using heuristics to reduce the search space of the solutions, and the third one uses an Evolutionary Algorithm to find the best answer. We analyze their complexity and solve the same problem of the Example 3.6 with each of them. The complete results of applying them on out test data is provided in the Section 3.3.

3.2.3.2 Top-Down Heuristic Search Algorithm

In the top-down search for the optimum combination of rectangles we start from the smallest single rectangle that covers all the irregular cells. Let n_{r_0} be the number of regular cells that are covered by this rectangle. We stop if $n_{r_0} = 0$. If not, we consider all the possible combinations of breaking this rectangle into two rectangles such that they together cover all the irregular cells, and also no irregular cell i covered by both rectangles. We call the number of the regular cells cover by the two rectangle in this step $n_{r_1^0}$ and $n_{r_1^1}$, and we choose the combination for which $n_{r_1^0} + n_{r_1^1}$ is minimized. We repeat this procedure for each of the new rectangles until there is no regular cells in any of the rectangles. The idea behind the heuristic of this method is to divide a rectangle only if it is needed, and then divide it to the minimum (i.e. 2) number of rectangles. The solution of this method for the Minimum Irregular Regions Problem (Definition 3.1) is presented in Algorithm 3.4. It is called Top-Down because we start with one rectangle and break it down to as many as required to satisfy the requirements of the Minimum Irregular Regions Problem.

Algorithm 3.4 Top-Down Search For Minimum Irregular Regions Problem

```
1: Create two queues named $C$ and $NC$. Initialize both to empty queues.
2: Enqueue the rectangle $\Gamma^{-1}(I)$ in $NC$.
3: If the queue $NC$ is empty, the set of the rectangles in $C$ is the answer. If not, go to the
   Step 4.
4: Dequeue one rectangle from the queue $NC$, call it $c$
     • Queue $c$ in $C$ if it does not cover any object of $R$.
     • If there is any object in $R$ that is covered by $c$, find the partitioning of $\Gamma(c)$ in two
       partitions (named $s_1$ and $s_2$) where the total number of objects in $R$ covered by
       $c_1 = \Gamma^{-1}(s_1)$ or $c_2 = \Gamma^{-1}(s_2)$ is minimized. Enqueue both rectangles $c_1$ and $c_2$ in
       $NC$.
5: Go to the Step 3.
```

3.2.3.3 Analysis of Top-Down Heuristic Search Algorithm

The Top-Down Search algorithm imposes a condition on the irregular regions to decrease the search space: it only searches for the solutions where each irregular cell (i.e. an object in P whose label are i) is covered by one rectangle only. However, this algorithm still evaluates *all* the possible ways of breaking down a rectangle into two in Step 4. This means that the algorithm should evaluate $2^{|I|}$ cases at the very beginning, up to $2^{|I-1|}$ cases in the next iteration and so on. In the worst case, we have to go through $2^{|I|} + 2^{|I-1|+\ldots+1}$ cases. Checking to see if one rectangle is not covering any regular cell (i.e. objects in P whose label are r) itself has the complexity order of $O(|R|)$. Algorithm 3.4 therefore has the complexity of $O\left(m \times 2^{\frac{n(n+1)}{2}}\right)$, where $m = |R|$ and $n = |I|$. This exponential order means that the algorithm is still impractical for large Is. In the C++ implementation of the algorithm compiled using VC11 and running on an Intel®core™i7 processor, the largest size of $|I|$ for which the solution could be found in order of few hours was $|I| = 32$. This means that the Storing Ranges of Exception method using the Top-Down Heuristic Search Algorithm cannot find the transition function for the CA defined in the Example 3.6 in less than few days. Section 3.3 provides the results only for the inputs who had less than 32 irregular cells in their transition from any layer of the image to the next.

3.2.3.4 Bottom-up Heuristic Search Algorithm

Algorithm 3.4 adds all the irregular cells to a large rectangle at once in the first step. In contrast, the Bottom-Up Heuristic Search starts from a rectangle that covers only one irregular cell, and then adds the rest of irregular cells one by one, either expanding an existing rectangle or creating a new one in each step. Moreover, the Bottom-Up Heuristic Search does not impose the condition that each irregular cell is covered by one rectangle only. The idea behind the algorithm is to avoid creating a new rectangle unless non of the existing ones can be expanded to cover the new object without covering any regular cell.

Algorithm 3.5 explains the Bottom-Up search. It works incredibly faster than the Algorithm 3.4 but it searches only a small section of the search space. As we will see in the Section 3.3 using this algorithm can find the answer to all the test data in an acceptable time and the results are still superior to of the first Storing Exceptions method explained in the Section 3.2.2.

Algorithm 3.5 Bottom-Up Search For Minimum Irregular Regions Problem

```
1: Enqueue all element in $I$ in an initially empty queue named $O$. Create two empty sets named $C$ and $C'$.
2: Dequeue one element from $O$ and name it $o_1$. Add $\Gamma^{-1}(o_1)$ to $C$.
3: If $O$ is empty, $C$ is the solution. If not, dequeue the next element from $O$ and name it $o_{next}$
4: For each rectangle $c\left\langle(x_c^{ul}, y_c^{ul}), (x_c^{lr}, y_c^{lr})\right\rangle$ in $C$, form $c'\left\langle(x_{c'}^{ul}, y_{c'}^{ul}), (x_{c'}^{lr}, y_{c'}^{lr})\right\rangle = \Gamma^{-1}(\Gamma(c) + o_{next})$, and put all the $c'$s in $C'$.
5: Remove from $C'$ any rectangle $c'$ that $\exists o_r \in R, c' \boxdot o_r$.
6: Of the remaining $c'$s in $C'$, select the one that minimizes $(x_{c'}^{ul} - x_c^{ul})^2 + (y_{c'}^{ul} - y_{c'}^{ul})^2 + (x_{c'}^{lr} - x_{c'}^{lr})^2 + (y_{c'}^{lr} - y_{c'}^{lr})^2$. Replace its corresponding $c\left\langle(x_c^{ul}, y_c^{ul}), (x_c^{lr}, y_c^{lr})\right\rangle$ in $C$ with $c'\left\langle(x_{c'}^{ul}, y_{c'}^{ul}), (x_{c'}^{lr}, y_{c'}^{lr})\right\rangle$.
7: Go to the Step 3.
```

3.2.3.5 Analysis of Bottom-Up Heuristic Search Algorithm

Algorithm 3.5 has an important behavior: it considers only one order of enqueuing I in O in Step 1. this means we are searching one of possible $|I|!$ sections of the whole search space, with the guarantee that the algorithm always finds a possibly not optimum solution (Lemma 3.1). However it doesn't mean that this algorithm is less successful that the Top-Down algorithm in finding close to optimum solutions. The reason is that unlike Algorithm 3.4, it does not limit the search to the cases where each object in I is covered by one rectangle only. Moreover, the same solution can be found by searching in many different paths so the chance of finding the optimum solution is considerably more than $\frac{1}{|I|!}$. The restriction of the search space is the compromise to make the Algorithm 3.5 find a solution in acceptable time, in order of seconds (instead of hours) on the same machine mentioned in Section 3.2.3.3 for the same test inputs. Step 3 of the Algorithm 3.5 iterates $|I|-1$ elements and each iteration in the worst case processes $|I|$ rectangles in Step 5, where processing each rectangle is equivalent of checking its coverage of $|R|$ objects. Step 6 processes $|I|$ in the worst case. Algorithm 3.5 therefore has the complexity of $O\left(n(nm+n)\right) = O(mn^2)$, where $m = |R|$ and $n = |I|$. This polynomial order makes the runtime is short enough to be practical for even large inputs. Example 3.10 demonstrates using this algorithm for the Storing Ranges of Exceptions method to find the transition function for a CA.

Example 3.10.

Assume the same problem of Example 3.8; i.e. finding the transition function that develops the pattern of the right-most figure in table 3.3 from the middle figure in the same table. Using the Storing Ranges of Exceptions *method that uses the Top-Down Heuristic Search to find the set of rectangles representing the irregular regions results in the transition function demonstrated in the Equation 3.43. Figure 3.8 depicts the five irregular regions the Algorithm 3.5 finds for the rule* $\langle 1,1,1,1,1,1,1,1,1\rangle \longrightarrow 1$.

$$f = \begin{cases} \langle 0,0,0,0,1,1,0,1,1 \rangle & \longrightarrow \quad 1 \\ \langle 0,0,0,1,1,1,1,1,1 \rangle & \longrightarrow \quad 1 \\ \langle 0,0,0,1,1,0,1,1,0 \rangle & \longrightarrow \quad 1 \\ \langle 0,1,1,0,1,1,0,1,1 \rangle & \longrightarrow \quad 1 \\ \langle 1,1,1,1,1,1,1,1,1 \rangle & \longrightarrow \quad 1 \\ \quad unless: & \\ \quad (5 \leq h_1 \leq 12 \wedge 9 \leq h_2 \leq 9) & \\ \vee (6 \leq h_1 \leq 11 \wedge 8 \leq h_2 \leq 10) & \\ \vee (8 \leq h_1 \leq 9 \wedge 7 \leq h_2 \leq 11) & \\ \vee (15 \leq h_1 \leq 16 \wedge 8 \leq h_2 \leq 10) & \\ \vee (16 \leq h_1 \leq 16 \wedge 6 \leq h_2 \leq 12) & \\ \langle 1,1,0,1,1,0,1,1,0 \rangle & \longrightarrow \quad 1 \\ \langle 0,1,1,0,1,1,0,0,0 \rangle & \longrightarrow \quad 1 \\ \langle 1,1,1,1,1,1,0,0,0 \rangle & \longrightarrow \quad 1 \\ \langle 1,1,0,1,1,0,0,0,0 \rangle & \longrightarrow \quad 1 \end{cases} \tag{3.43}$$

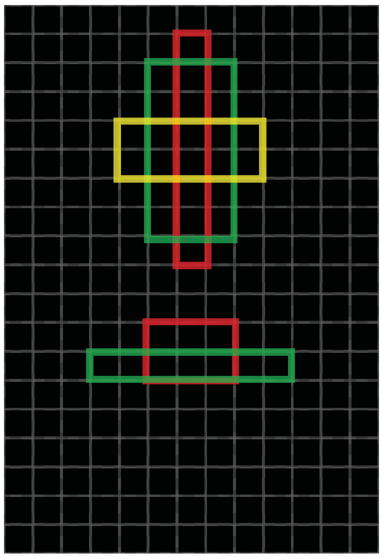


Figure 3.8: The irregular regions found by the Algorithm 3.5 for the 5th rule in the transition function of Equation 3.43

3.2.3.6 Using Evolutionary Algorithm

The Minimum Irregular Regions Problem is an NP-Complete problem that is almost impossible to be solved using an exhaustive search for the CAs containing more than few cells. Both Top-Down and Bottom-Up Heuristic Searches impose limiting conditions on the search space to approximate the original problem and find close-to-optimum solutions. On the other hand, evolutionary algorithms are known to be very effective tools to find the solution in large search spaces, where other heuristic searches often fail to find the solution in acceptable time. In this section we present an evolutionary algorithm to find the solution to the Minimum Irregular Regions Problem. For the ease of future references we call this evolutionary algorithm *EAMIRP*: the Evolutionary Algorithm for the Minimum Irregular Regions Problem.

- **Representation**

  The individuals in EMIRP should represent solutions to the Minimum Irregular Regions Problem. As defined in the Definition 3.1 each individual should be a set of rectangles. The same definition defines each rectangle as a pair of integer values in P, therefor each individual in EAMIRP is a set of vectors of four integers. To be consistent with the notation of Definition 3.1 we name this set C. $C(i)$ will denote the set of rectangles for the individual i, and $c_n(i)$ denotes the n^th rectangle in $C(i)$. The lower bound of the size of C is always 1 and there is no upper bound. The size of individual i is defined as the number of vectors in $C(i)$.

- **Initialization**

  EAMIRP starts with a fixed number of individuals in the first generation, and the population size stays constant in course of evolution. All the individuals are initialized to $\Gamma^{-1}(I)$. This means the first generation in EAMIRP contains identical copies of the same genotype.

- **Penalty Function**

  EAMIRP uses penalty instead of fitness function. The penalty assigned to each individual is calculated according to the Equation 3.44, where $Penalty(i)$ is the penalty of the given individual i, $Size(i)$ is the size of the individual i as defined in the Representation and d_1 and d_2 are the size of the dimensions D_1 and D_2 of the CA.

$$Penalty(i) = Size(i) \times \left(n_r(i) + \sqrt{\frac{d_1 + d_2}{2}} \right)^2 \tag{3.44}$$

  $n_r(i)$ in the Equation 3.44 is the number of times a member of R is covered by a rectangle in $C(i)$, and is defined in the Equation 3.45.

$$n_r(i) = \sum_{m=0}^{|R|} \sum_{n=0}^{Size(i)} W(m,n,i) \tag{3.45}$$

where $W(m,n,i)$ is defined in the Equation 3.46

$$W(m,n,i) = \begin{cases} 1, & c_n(i) \boxdot o_r^m \\ 0, & otherwise \end{cases} \tag{3.46}$$

o_r^m is the m^th element of the set R in the the Definition 3.1.

EAMIRP is searching for the *smallest* set of irregular regions that cover *zero* element in R. The penalty function of the Equation 3.44 therefore is reflecting this requirement. It has two main parts, one representing the effect of number of irregular regions ($Size(i)$) and another one represents the effect of objects in R located inside the irregular regions. The latter participates in its square form because there exists a harder condition on the number of objects in R covered in the solution. It also gets added to a constant (a function of size of the CA, therefore constant during the algorithm) because we do not want to stop evolution as soon as we found a solution that does not cover any element in R. The penalty will be calculated as zero for such solution no matter what $Size(i)$ is. We should avoid such termination point so we can keep looking for potentially smaller sets. The penalty function does not take into account the number of elements in I that are not covered by any member of C, because as we will see later, EAMIRP guarantees that each individual always covers *all* the elements in I.

- **Parent Selection**

  EAMIRP uses a tournament parent selection with the tournament size set to 3. Three individuals are selected using a random uniform probability density function. The selection does not prevent an individual from being selected more than once in one tournament. The individual with the lowest fitness in the tournament is selected as a parent. In case of a tie, the individual with the lowest index in the tournament pool is selected as the parent. If the size does not break the tie the individual who was selected to the tournament first is selected as the parent. We repeat this for a fixed

N_c number of times to fill in the parent pool.

- **Reproduction**

  EAMIRP uses asexual reproduction and each parent generates exactly one child using mutation. There are two types of mutation: Merge and Split. Only one type of mutation is applied on a parent to create a child. The probability of each mutation type is presented in the Equations 3.47 and 3.48.

$$P(\mu_{merge}) = \frac{Size(i)}{|I|} \tag{3.47}$$

$$P(\mu_{split}) = 1 - P(\mu_{merge}) \tag{3.48}$$

  as it can be observed from the Equation 3.47, when an individual has one irregular per each object of I, $Size(i) = |I|$ and a merge mutation will be applied on the individual. On the other hand, when there is only one irregular region that covers all the elements of I, $Size(i) = 1$ and the split mutation has a very high chance (close to %100, when I grows large) to be applied. The merge and split mutations are explained in the Algorithm 3.6 and 3.7 respectively. It can be observed in those algorithms that it is guaranteed that all the members of I are always covered by one or more member of C for any individual at any time. All the created child are added to the population.

Algorithm 3.6 The Mutation Merge in the Evolutionary Algorithm Search For Minimum Irregular Regions Problem

```
1: Stop if $|C| = 1$.
2: Choose two random member $c_1$ and $c_2$ from $C$ with replacement and uniform distribution. The same member can be selected twice.
3: Remove $c_1$ and $c_2$ from $C$.
4: Add $\Gamma^{-1}(\Gamma(c_1) \cup \Gamma(c_2))$ to $C$.
```

- **Survival Selection**

  EAMIRP uses elitism, so a fixed number of the best individuals (i.e. the individuals with the lowest penalty before the children were added) are directly transfered to

Algorithm 3.7 The Mutation Split in the Evolutionary Algorithm Search For Minimum Irregular Regions Problem

```
0: Let $C' = \{c\langle(x_c^{ul}, y_c^{ul}), (x_c^{lr}, y_c^{lr})\rangle \in C \mid x_c^{ul} = x_c^{lr} \Longrightarrow y_c^{ul} \neq y_c^{lr}\}$. We know $C' \neq \phi$
   because if otherwise, $Size(i) = |I|$ and $P(\mu_{split})$ would be zero.
1: Select a random member of $C'$ with uniform probability and call it $c$.
2: If $x_c^{ul} = x_c^{lr}$ go to the Step 5.
3: If $y_c^{ul} = y_c^{lr}$ go to the Step 6.
4: With an equal chance, go to either Step 5 or Step 6.
5: Perform the Vertical Split defined in the Algorithm 3.8 on $c$ to produce two new rect-
   angles $c_1$ and $c_2$. Go to the Step 7.
6: Perform the Horizontal Split defined in the Algorithm 3.9 on $c$ to produce two new
   rectangles $c_1$ and $c_2$.
7: Remove $c$ from $C$. If $Gamma(c_i)_{i=1,2} \neq \phi$ then add $c_i$ to $C$.
```

Algorithm 3.8 The Vertical version of Mutation Split in the Evolutionary Algorithm Search For Minimum Irregular Regions Problem

```
1: Pick a random $o_t \langle i, (x_t, y_t)\rangle$ from $\Gamma(c)$ (note that all the members in $\Gamma(c)$ have $Label =$
   $i$).
2: Set $T_1 = \{o\langle i, (x, y)\rangle \mid x \leq x_t\}$ and $T_2 = \Gamma(c) - T_1$.
3: Set $c_1 = \Gamma^{-1}(T_1)$ and $c_2 = \Gamma^{-1}(T_2)$.
```

Algorithm 3.9 The Vertical version of Mutation Split in the Evolutionary Algorithm Search For Minimum Irregular Regions Problem

```
1: Pick a random $o_t \langle i, (x_t, y_t)\rangle$ from $\Gamma(c)$ (note that all the members in $\Gamma(c)$ have $Label =$
   $i$).
2: Set $T_1 = \{o\langle i, (x, y)\rangle \mid y \leq y_t\}$ and $T_2 = \Gamma(c) - T_1$.
3: Set $c_1 = \Gamma^{-1}(T_1)$ and $c_2 = \Gamma^{-1}(T_2)$.
```

the next generation. For filling in the population of the next generation EAMIRP uses fitness proportional selection. Each individual is assigned the fitness equal to $Fitness(i) = \max_{j=0}^{N_p} Penalty(j) - Penalty(i)$, and $N_p - N_e$ number of individuals are selected once at a time (N_p is the population size before adding the children and N_e is the number of elites in the algorithm). In each time the chance of each individual for being transfered to the next generation is $\frac{Fitness(i)}{\sum_{j=0}^{N_p+N_c} Fitness(j)}$. EAMIRP does not prevent an individual to be copied to the population of next generation more than once.

- **Termination Criteria**

  EAMIRP stops when it exhausts a fixed number of generation or if there is no decreasing of the penalty of the population's lowest penalty over a certain number of generations. The individual with the lowest penalty in the last generation is returned as the solution. The solution is checked to have no element in R covered by any element of C.

The evolutionary algorithm of EAMIRP is illustrated in Figure 3.9. Example 3.11 demonstrates using EAMIRP with the parameters listed in the Table 3.6 for the Storing Ranges of Exceptions method to find the transition function for a CA.

Example 3.11.
Using EAMIRP with the parameters listed in the Table 3.6 to solve the same problem of Example 3.8 will result in the transition function presented the Equation 3.49. This solution is found after only 22 generations since the input problem is rather simple. Figure 3.10 depicts the five irregular regions EAMIRP finds for the rule $\langle 1,1,1,1,1,1,1,1,1\rangle \longrightarrow 1$. *As it can be observed, the solution found by the EAMIRP is very similar to of the Bottom-Up Heuristic Search algorithm. They both have the same number of rectangles, although the latter has been searching a subset of the search space. The performance of both methods is measured on a large test data in Section 3.3.*

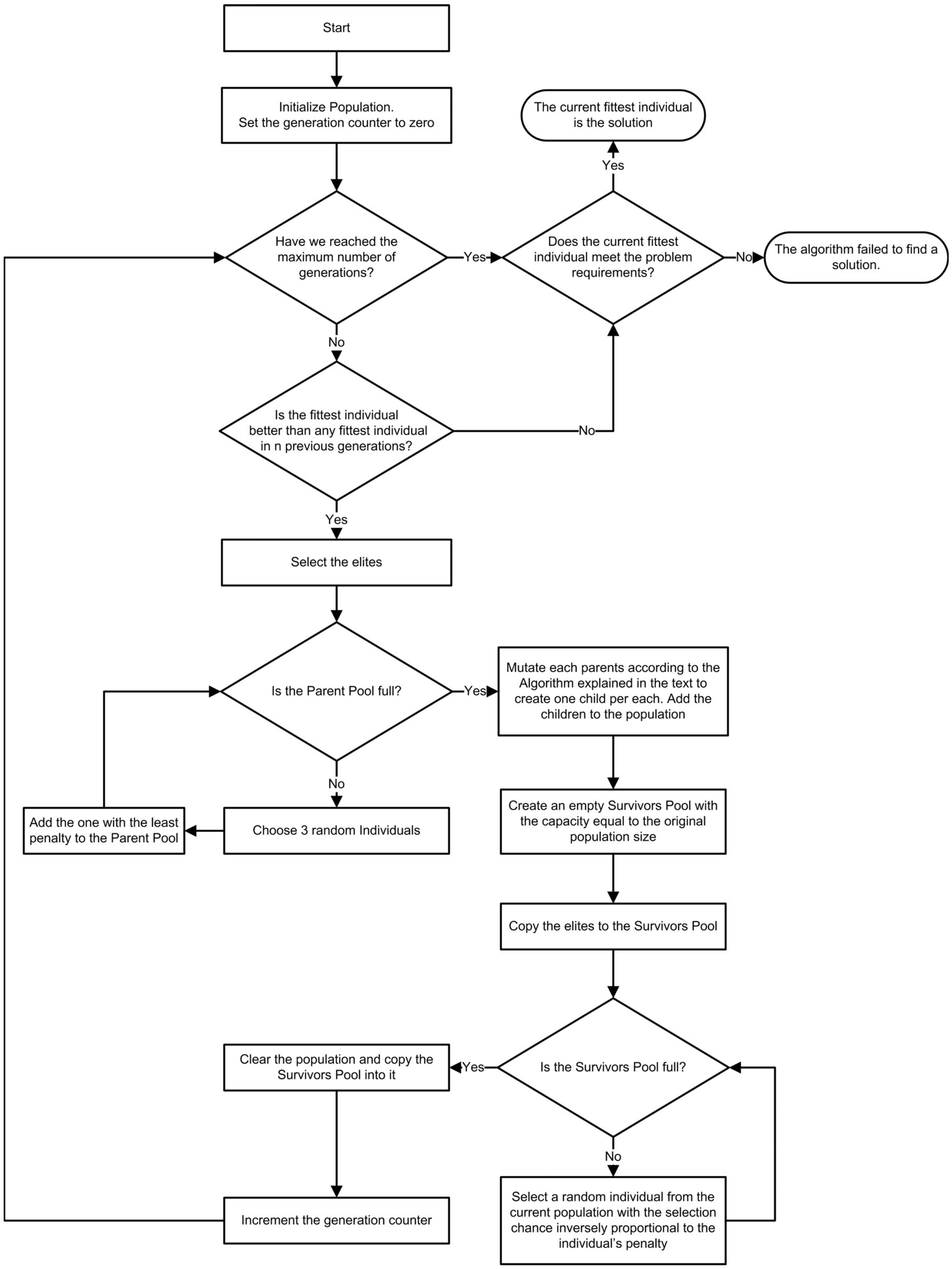

Start
Initialize Population. Set the generation counter to zero
Have we reached the maximum number of generations?
Yes
Does the current fittest individual meet the problem requirements?
Yes
The current fittest individual is the solution
No
The algorithm failed to find a solution.
No
Is the fittest individual better than any fittest individual in n previous generations?
No
Yes
Select the elites
Is the Parent Pool full?
Yes
Mutate each parents according to the Algorithm explained in the text to create one child per each. Add the children to the population
No
Choose 3 random Individuals
Add the one with the least penalty to the Parent Pool
Create an empty Survivors Pool with the capacity equal to the original population size
Copy the elites to the Survivors Pool
Is the Survivors Pool full?
Yes
Clear the population and copy the Survivors Pool into it
No
Select a random individual from the current population with the selection chance inversely proportional to the individual's penalty
Increment the generation counter

Figure 3.9: The flowchart of the evolutionary algorithm used in EAMIRP.

$$f = \begin{cases} \langle 0,0,0,0,1,1,0,1,1\rangle & \longrightarrow \quad 1 \\ \\ \langle 0,0,0,1,1,1,1,1,1\rangle & \longrightarrow \quad 1 \\ \\ \langle 0,0,0,1,1,0,1,1,0\rangle & \longrightarrow \quad 1 \\ \\ \langle 0,1,1,0,1,1,0,1,1\rangle & \longrightarrow \quad 1 \\ \\ \langle 1,1,1,1,1,1,1,1,1\rangle & \longrightarrow \quad 1 \\ \quad \mathit{unless}: \\ \quad (16 \le h_1 \le 16 \wedge 6 \le h_2 \le 12) \\ \vee (5 \le h_1 \le 12 \wedge 9 \le h_2 \le 9) \\ \vee (8 \le h_1 \le 9 \wedge 7 \le h_2 \le 11) \\ \vee (15 \le h_1 \le 16 \wedge 8 \le h_2 \le 10) \\ \vee (6 \le h_1 \le 11 \wedge 8 \le h_2 \le 10) \\ \\ \langle 1,1,0,1,1,0,1,1,0\rangle & \longrightarrow \quad 1 \\ \langle 0,1,1,0,1,1,0,0,0\rangle & \longrightarrow \quad 1 \\ \langle 1,1,1,1,1,1,0,0,0\rangle & \longrightarrow \quad 1 \\ \langle 1,1,0,1,1,0,0,0,0\rangle & \longrightarrow \quad 1 \end{cases} \tag{3.49}$$

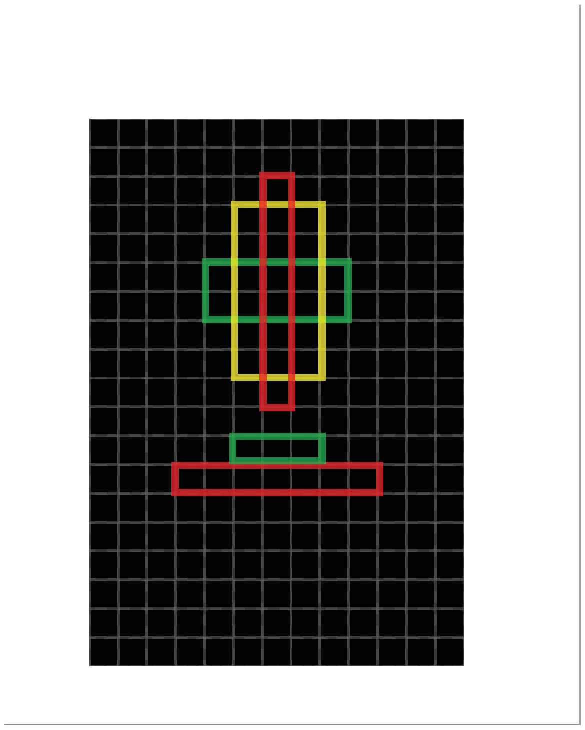


Figure 3.10: The irregular regions found by EAMIRP for the 5th rule in the transition function of Equation 3.49

Table 3.6: Parameters of the Evolutionary Algorithm in EAMIRP.

Parameter	Value
Population size	200
Maximum Generation	2000
Number of steady generations before stopping the Algorithm	100
Parent Pool Size	100
Tournament Size	3
Number of Elites	2

3.3 Experiments and Results

3.3.1 Experiment Setup

In this section we describe the experiment setups for measuring and comparing the performance of the methods presented in the Section 3.2. As mentioned in the Section 3.1.4, the problem is to find the neighborhood and the transition function for two or more consequent configurations of a CA of known size. All methods of Section 3.2 try to find and express the transition function as a set of *if-then* rules explained in the section 3.1.1. The number of rules and the storage size of the transition function (i.e. total amount of memory in bits required to store the transition function) is presented for the results of each method. The methods explained in the Section 3.2.1 will find neighborhoods of variable sizes, while the rest of the methods have fixed neighborhood size. The neighborhood size is therefore compared only for the former methods. To obtain the configurations of the CAs at time t ($t \in \mathbb{Z}$) we use the pattern resulting from extracting the t$^{\text{th}}$ bit of an 8-bit, 256-level gray scale image as explained in Section 3.1.4, Figure 3.6 and Table 3.1. We call the binary pattern generated from i$^{\text{th}}$ bit of the image *layer* i of the image. Layer 0 will be the binary pattern of the most significant bit of the value of each pixel in the image. The test data is a subset of the face portraits from [19]. We select the images in the first two folders in the http://vision.ucsd.edu/extyaleb/CroppedYaleBZip/CroppedYale.zip package, as presented in the appendix A.1. There are 64 images in each folder, making it a total of 128 images.

To measure the scalability of the methods we create two sets of each of the above image types, one containing 20×25 pixel images (Table A.1) and the other containing 40×50 pixel images (Table A.2). The face images are resized from the original image using Adobe® Photoshop® CS4's resize tool and Bicubic re-sampling of the original image.

Having eight consequent configurations of the CA (one per each layer of the image) enables us to study the effects of adding memory to CA as explained in Section 3.1.2. In the following experiments we search for the transition functions for generating each of the image layers such as layer i from its j immediately previous layers ($1 \leq j \leq i-1$). We will form tables of the format similar to the Table 3.7 for this purpose, where $f_{j-i,j-i+1,\ldots,j\rightarrow j+1}$ means the transition function that generates layer $j+1$ from the last i layers, and f^p

Table 3.7: The template of the result tables

	Make layer 1	Make layer 2	Make layer 3	Make layer 4	Make layer 5	Make layer 6	Make layer 7
Start from layer 0	$f^p_{0\to1}$	$f^p_{0,1\to2}$	$f^p_{0,1,2\to3}$	$f^p_{0,\ldots,3\to4}$	$f^p_{0,\ldots,4\to5}$	$f^p_{0,\ldots,5\to6}$	$f^p_{0,\ldots,6\to7}$
Start from layer 1		$f^p_{1\to2}$	$f^p_{1,2\to3}$	$f^p_{1,2,3\to4}$	$f^p_{1,\ldots,4\to5}$	$f^p_{1,\ldots,5\to6}$	$f^p_{1,\ldots,6\to7}$
Start from layer 2			$f^p_{2\to3}$	$f^p_{2,3\to4}$	$f^p_{2,3,4\to5}$	$f^p_{2,\ldots,5\to6}$	$f^p_{2,\ldots,6\to7}$
Start from layer 3				$f^p_{3\to4}$	$f^p_{3,4\to5}$	$f^p_{3,4,5\to6}$	$f^p_{3,\ldots,6\to7}$
Start from layer 4					$f^p_{4\to5}$	$f^p_{4,5\to6}$	$f^p_{4,5,6\to7}$
Start from layer 5						$f^p_{5\to6}$	$f^p_{5,6\to7}$
Start from layer 6							$f^p_{6\to7}$

denotes the property p of such transition function. p can be any of the following properties, depending on the form of the transition function f.

- The neighborhood size (for the expanding neighborhood method in Algorithm 3.1 only)
- Number of rules in the transition function (expressed as a rule set)
- The storage size of the transition function, defined as the amount of memory required to store the transition function (in bits)
- The time that was spent to find the transition function (in milliseconds)
- The number of times the specific algorithm failed to find the transition function in the set of 128 test data (different reasons of failure are discussed later in their own section). If this number is not presented it means that the algorithm was successful in finding the transition function for all the test data.

We do not list the detailed results or running each algorithm on each image in all the test data sets, as that will require hundreds of pages. Each table has the average of the specific property of the transition function (e.g. neighborhood size) for all the 128 images of a set. The C++ source code of program and the test data sets are available on the public SVN repository `https://cmeasure3.googlecode.com/svn/trunk/cmeasure3` . Although the code is originally compiled on a Microsoft Windows 7 PC with Microsoft Visual Studio 2012 and the VC++11 compiler, it has no dependency on any specific OS or compiler (it has been compiled successfully on a Linux machine with GCC as well). The program also runs in multi thread mode in some computationally expensive modules, if multi-threading is supported by the host machine and the compiler.[1]

Each table in the form of Table 3.7 includes 28 pieces of data. Some cells are blank because it doesn't make sense to find the past configuration of the CA from its future configuration, or to find its current configuration from it current configuration. Note that the first four tables provide the *average* of 128 cases for each of the 28 transition functions, while the last type of table (The number of times the algorithm failed) shows the absolute number of times the algorithm failed. The reason for failure of the algorithm depends on the specific algorithm, listed below:

- The *Expanding Neighborhood* method (Algorithm 3.1 in Section 3.2.1) might fail if there remains at least one conflict in the Step 3 of the algorithm no matter how large the neighborhood is extended. We remember that the current state of any cell outside the CA was assumed to be a constant values in S. No matter what this value is, it has the chance to be identical to the current configuration of the part of neighborhood inside the CA for the specific cell that is causing the conflict.

- The *Storing Individual Exception* method (Algorithm 3.2 in Section 3.2.2.2) will never fail.

- The *Storing Ranges of Values to Store Exceptions* method (Section 3.2.3) might fail for different reasons depending on the specific search algorithm used to solve the Minimum Irregular Regions Problem:

  - The *Top-Down* search (Algorithm 3.4) might fail if the required resources (either time or memory) exceed the available computing power. In our implementation the algorithm is forced to quit whenever in Step 4.2 of the algorithm the number of object in R that are covered by c are more than 32. Although a 64-bit machine

[1] I need to thank my friend Marc-Antoine Chabot, who helped me get my results faster by lending me his computer's CPU time.

can handle the cases where up to 64 objects in R are covered by c, the exponential order of the algorithm prevents us to pass that step in less than few hours on an Intel®core$^{\text{TM}}$i7 machine when that number is higher than 32.

– The *Bottom-Up* search (Algorithm 3.5) never fails (in price of searching a small section of the search space).

– The *Evolutionary* search (Figure 3.9) might fail if trapped in a local optima point. It is possible to get trapped in such local optima because we do not know the minimum possible penalty of the individuals in advance, and also because we stop the algorithm if there has been no decrease in the penalty for a fixed number of generations.

3.3.2 Results - Expanding Neighborhood

We remember from Example 3.12 that the order according to which the neighbors are evaluated for their importance can effect the final neighborhood size. Moreover, two initially distinguishable rules can become identical after discarding enough elements from the neighborhood. As a result, the order of evaluating the neighbors plays a role in the neighborhood size, number of the rules in the transition function and the most important, in the storage size of the transition function. For all the methods in this section we store only the rules whose output is 1, therefore the next state of a cell will be 0 if no rule in the transition function matches its neighborhood.

In this section we run the Algorithm 3.1 on different set of inputs obtained from different sets of images as explained earlier. In each following sections we modify the Step 4 of the Algorithm 3.1 to try different orders on the test data sets. We measure the following parameters of the resulted neighborhood and transition function in each case:

- Neighborhood Size: the number of elements in N.
- Number of Rule: the number of rules in the transition function.
- Storage Size: the amount of memory in bits required to store the transition function on a binary machine.

- Calculation Time: the time in milliseconds required to find the transition function. To get a better comparison of the effect of each removing order, we have measured the time only for step 4 of the algorithm 3.1, as the first 3 steps are constant for all.

- Number of failures: the number of times that the Algorithm 3.1 could not find the transition function. It happens when there remains an un-resolved conflict no matter how large the neighborhood is increased in the Step 3 of the algorithm.

The table showing the last property - the number of times the algorithm failed - is the same for all methods explained in this section. The reason is that the methods here differ only in Step 4 of the Algorithm 3.1, while the failure happens in the Step 3, a common step for all the methods in this section. We provide that information in one single table for all the methods in Table 3.8.

Of the mentioned properties of the results, the *Storage Size* is the most important one, since after all it is the transition function that needs to be stored in each cell. When implemented on a stand-alone hardware, a small change in the size of transition function can result in magnificent difference in total memory requirement of the device. For saving memory, the transition function stores only the rules whose output are 1. This was discussed earlier in the Section 3.1.4. The calculation time gives us only a rough estimation of the run time of each algorithm and not a very precise measure of comparison. This is because the run time depends not only on the complexity of the algorithm but the efficiency of the implementation as well. The better measure for comparing the cost of the algorithm is the complexity analysis provided for each of the algorithms in Section 3.2.

For each order of removing the neighbors, each of the above three properties are measured for 128 images in each of the test data sets as explained in the Section 3.3.1. The average in each run is presented in the following tables, e.g. the number in the row L_i and column L_j in a table that shows the neighborhood sizes, means that the average neighborhood size for a transition function that creates the layer j of the image from layers $i, i+1, \ldots, j-1$. The CA is memory less if $j = i+1$, and has a memory of the last $j-i-1$ steps otherwise.

Table 3.8: The number of times the *Expanding Neighborhood* method fails to find the transition function.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	1	0	0	0	0	0	0
L_1		0	0	0	0	0	0
L_2			0	0	0	0	0
L_3				0	0	0	0
L_4					0	0	0
L_5						0	0
L_6							0

Table 3.9: The average of neighborhood sizes resulting from *remove oldest first* (the current existing method).

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	35.05	18.55	14.16	12.37	9.25	7.24	5.41
L_1		22.76	15.36	12.88	9.38	7.30	5.43
L_2			22.30	14.74	9.98	7.52	5.70
L_3				22.45	11.87	8.23	6.05
L_4					18.29	10.01	6.80
L_5						15.97	8.67
L_6							15.38

3.3.2.1 Removing Oldest Elements First

Tables 3.9 to 3.12 provide the result for each of the four properties listed earlier in this section, when the step 4 of the algorithm 3.1 starts from the element in the neighborhood who has been added prior to the others. We use the phrase *remove oldest first* to refer to this order.

Table 3.10: The average of number of rules in the transition function resulting from *remove oldest first* (the current existing method).

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	191.62	219.68	243.1	237.06	237.89	249.17	248.96
L_1		219.13	243.42	236.94	237.96	249.17	248.96
L_2			243.21	236.87	238	249.1	249
L_3				235.78	237.66	249.31	248.8
L_4					236.56	249.21	248.73
L_5						248.59	248.85
L_6							247.87

Table 3.11: The average number of bits required to store the transition function resulting from *remove oldest first* (the current existing method).

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	6836.36	8222.92	10729.99	11440.78	10703.55	10786.13	9420.47
L_1		4963.06	7759.39	8939.09	8688.78	9066.48	8098.50
L_2			5604.52	6834.86	6932.39	7471.50	7081.88
L_3				5180.42	5506.77	6142.01	6021.56
L_4					4252.41	4980.23	5070.23
L_5						3969.61	4315.77
L_6							3811.98

Table 3.12: The average time in milliseconds required to find the transition function resulting from *remove oldest first* (the current existing method).

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	2424.89	1026.94	2125.71	3245.53	1784.62	495.78	260.46
L_1		620.06	1330.89	2057.34	1242.88	381.15	202.57
L_2			1073.99	1567.29	940.18	299.84	166.59
L_3				1193.06	788.56	233.62	140.37
L_4					567.36	195.6	113.34
L_5						190.95	118.46
L_6							154.41

Table 3.13: The average of neighborhood sizes resulting from the *remove newest first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	28.63	16.99	13.01	11.60	9.18	7.08	5.49
L_1		20.35	14.02	11.91	9.30	7.13	5.52
L_2			19.84	13.38	9.62	7.31	5.77
L_3				19.84	11.29	7.94	6.09
L_4					17.03	9.55	6.84
L_5						15.22	8.41
L_6							14.88

Table 3.14: The average of number of rules in the transition function resulting from the *remove newest first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	167.17	199.04	231.64	234.05	236.17	247.71	248.22
L_1		199.44	231.61	233.71	236.22	247.72	248.17
L_2			232.08	232.57	236.16	247.61	248.26
L_3				230.95	236.17	247.78	248.33
L_4					235.82	247.94	248.28
L_5						247.31	248.25
L_6							247.78

3.3.2.2 Removing Newest Elements First

In order to observe the effect or picking the oldest element of the neighborhood in Step 4 of the algorithm 3.1, we also designed a set of experiments to show the results if the newest element in the neighborhood is examined for removal first. We call this order *remove newest first* and provide its results in the Tables 3.13 to 3.16.

Table 3.15: The average number of bits required to store the transition function resulting from the *remove newest first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	4879.87	6841.09	9569.63	10518.25	10521.48	10455.33	9512.40
L_1		4071.60	6825.09	8084.04	8528.91	8771.60	8197.50
L_2			4749.88	6021.44	6622.34	7206.03	7147.30
L_3				4462.66	5207.86	5874.73	6040.47
L_4					3963.23	4725.61	5090.93
L_5						3762.22	4177.63
L_6							3686.34

Table 3.16: The average time in milliseconds required to find the transition function resulting from the *remove newest first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	2333.21	916.23	1888.87	3057.88	1909.43	533.39	263.35
L_1		520.79	1184.62	1944	1240.23	390.93	208.01
L_2			924.08	1421.11	942.42	304.78	171.6
L_3				1033.4	778.12	227.29	135.66
L_4					523.46	200.52	105.19
L_5						197.85	110.79
L_6							155.56

Table 3.17: The average of neighborhood sizes resulting from the *least dynamic information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	36.25	20.14	14.87	12.45	9.96	7.54	5.66
L_1		23.88	15.95	13.13	9.95	7.45	5.70
L_2			22.51	14.80	10.46	7.67	5.86
L_3				22.07	12.09	8.41	6.13
L_4					18.05	9.77	6.89
L_5						15.59	8.55
L_6							15.16

3.3.2.3 Removing Least Gainful Elements First, Dynamically Updating Gains

The method used here is very similar to the method of Section 3.3.2.5 , instead the information gain of each neighbor is updated once a neighbor is removed, and the remaining neighbors are sorted from the least information gain to the highest after each removal. We call this the *least dynamic information gain first*, and show its results in the Tables 3.17 to 3.21.

Table 3.18: The average of number of rules in the transition function resulting from the *least dynamic information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	188.66	216.57	242.32	236.25	237.28	248.69	248.76
L_1		216.08	244.46	236.82	237.61	248.62	248.85
L_2			243.47	237.19	237.67	248.32	249.07
L_3				234.39	237.5	248.99	248.92
L_4					236.49	248.47	248.6
L_5						248.37	248.53
L_6							248.11

Table 3.19: The average number of bits required to store the transition function resulting from the *least dynamic information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	6831.57	8813.39	11348.37	11440.56	11401.09	11203.88	9831.17
L_1		5073.52	8202.69	9071.27	9146.84	9215.00	8488.31
L_2			5745.51	6849.38	7223.37	7577.88	7277.85
L_3				5034.20	5576.80	6262.83	6089.25
L_4					4194.04	4838.88	5133.33
L_5						3871.08	4251.94
L_6							3761.94

Table 3.20: The average time in milliseconds required to find the transition function resulting from the *least dynamic information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	449204.32	125534.84	738182.84	1256526.15	591368.1	61930.46	14312.14
L_1		31027.88	292301.82	602187.15	331843.85	34827.56	8993.65
L_2			128069.76	290467.5	189476.23	21009.07	5784.33
L_3				111144.93	103457.94	11959.54	3519.39
L_4					34932.64	5835.62	1963.09
L_5						2374.13	1352.85
L_6							1019.25

Table 3.21: The average of neighborhood sizes resulting from the *most dynamic information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	40.96	21.09	15.88	13.94	10.02	7.45	5.66
L_1		24.62	16.65	14.05	10.06	7.57	5.66
L_2			23.37	15.84	10.71	7.64	5.88
L_3				23.26	12.46	8.40	6.29
L_4					18.66	9.91	7.03
L_5						15.75	8.77
L_6							15.27

Table 3.22: The average of number of rules in the transition function resulting from the *most dynamic information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	201.91	224.06	245.48	238.66	237.98	249.22	249.12
L_1		222.42	244.96	238.05	238.06	249.25	249.14
L_2			244.21	238.2	238.46	249.21	249.43
L_3				236.29	238.3	249.23	249.27
L_4					237.45	249.33	249.21
L_5						249.06	249.2
L_6							248.31

3.3.2.4 Removing Most Gainful Elements First, Dynamically Updating Gains

Once again to see the effect of re-sorting the neighbors according to their information gain in step 4 of the Algorithm 3.1, we repeat the experiment while the neighbors are dynamically sorted similar to of the Section 3.3.2.6, but in a reverse order from the neighbor with the most information gain to the one with the least. Similar to the Section subsubsec:Removeleastgainfirst-dynamic, the neighbors are re-sorted after each removal. The results for this method - called the *most dynamic information gain first* order - are presented in the Tables 3.17 to 3.20.

3.3.2.5 Removing Least Gainful Elements First, No Update in Gains

Tables 3.25 to 3.29 provide the features of the transition function resulting from the Algorithm 3.1 when in the Step 4 we sort the neighbors from the least information gain to the most, and we *do not* update the information gain of the neighbors after each removing of the neighbors. We call this order *least static information gain first*.

Table 3.23: The average number of bits required to store the transition function resulting from the *most dynamic information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	8285.14	9548.75	12317.23	12954.25	11531.02	11106.61	9855.51
L_1		5419.03	8554.55	9814.10	9274.72	9415.16	8442.00
L_2			5959.74	7375.78	7419.42	7599.38	7333.91
L_3				5378.53	5779.80	6265.73	6262.47
L_4					4343.71	4936.84	5253.59
L_5						3922.20	4369.69
L_6							3790.66

Table 3.24: The average time in milliseconds required to find the transition function resulting from the *most dynamic information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	248904.85	122186.18	666788.57	1304033.45	631127.9	61760.82	14718.93
L_1		30453.22	277465.6	607790.35	358116.17	34799.41	9197.14
L_2			115653.78	296991.67	208981.89	21247.06	5953.62
L_3				109789.31	118018.25	11584.9	3583.86
L_4					38150.36	5691.71	1956.15
L_5						2413.29	1394.87
L_6							1023.55

Table 3.25: The average of neighborhood sizes resulting from the *least static information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	28.13	16.76	12.65	11.02	8.57	7.02	5.45
L_1		20.09	13.57	11.41	8.70	7.04	5.52
L_2			19.63	13.02	9.24	7.23	5.66
L_3				20.08	11.01	7.92	5.93
L_4					17.13	9.50	6.73
L_5						15.36	8.43
L_6							15.02

Table 3.26: The average of number of rules in the transition function resulting from the *least static information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	167.52	201.22	233.06	233.92	236.38	248.21	248.75
L_1		201.34	232.91	233.38	236.38	248.21	248.73
L_2			231.72	234.07	236.6	248.03	248.75
L_3				230.98	236.39	248.22	248.53
L_4					235.79	248.23	248.6
L_5						248.23	248.57
L_6							248.03

Table 3.27: The average number of bits required to store the transition function resulting from the *least static information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	4826.54	6825.33	9148.99	10028.94	9849.80	10385.72	9460.94
L_1		4051.73	6500.09	7756.31	7996.75	8681.45	8214.33
L_2			4723.75	5950.53	6388.97	7128.91	7031.05
L_3				4531.18	5092.25	5869.76	5884.75
L_4					3986.15	4701.78	5016.12
L_5						3811.09	4190.00
L_6							3725.97

Table 3.28: The average time in milliseconds required to find the transition function resulting from the *least static information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	4721.52	2360.12	5719.63	10076.58	6983.34	2693.05	1709.39
L_1		983.57	2829.57	5494.46	4108.04	1748.75	1133.91
L_2			1619.91	3151.59	2582.32	1134.32	828.93
L_3				1711.61	1633.69	698.39	533.97
L_4					819.22	462.57	334.82
L_5						326.63	276.09
L_6							265.23

Table 3.29: The average of neighborhood sizes resulting from the *most static information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	34.90	18.61	13.89	12.12	9.08	7.12	5.46
L_1		22.74	15.18	12.53	9.25	7.18	5.52
L_2			21.95	14.41	9.80	7.42	5.68
L_3				21.55	11.52	8.08	5.99
L_4					17.89	9.77	6.75
L_5						15.50	8.52
L_6							15.08

Table 3.30: The average of number of rules in the transition function resulting from the *most static information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	191.28	218.03	241.8	237.51	237.67	249.07	248.94
L_1		218.41	241.46	237.42	237.82	249.08	248.95
L_2			242.48	237.51	237.78	249.07	249.04
L_3				235.38	237.71	249.28	248.81
L_4					236.7	249.22	248.98
L_5						248.63	248.82
L_6							248.03

3.3.2.6 Removing Most Gainful Elements First, No Update in Gains

To see the effect of sorting the neighbors according to their information gain and have a better understanding of the results of Section 3.3.2.5, we also measure the results when we sort the neighbors from the one with the most information gain to the one with the least in step 4 of the Algorithm3.1. The neighbors are not re-sorted according to their updated information gain after each neighborhood removal. Tables 3.29 to 3.32 present the results in this case, named *most static information gain first* order here.

3.3.2.7 Removing Row by Row, Top Left First

We mentioned before in Section 3.2.1.2 that we expect the physically close cells to have similar properties such as importance, and we suggested to examine removing the unimportant neighbors according to their location in the CA. In this section we try out this idea. The neighbors -who initially form a square at the end of Step 3 of the Algorithm 3.1 - are sorted row by row, from the top left neighbor to the bottom right neighbor. We call it the

Table 3.31: The average number of bits required to store the transition function resulting from the *most static information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	6756.98	8179.52	10488.73	11233.69	10468.40	10591.36	9504.47
L_1		4932.92	7597.91	8722.24	8540.53	8904.02	8241.80
L_2			5469.24	6702.91	6790.66	7369.22	7065.23
L_3				4962.94	5339.00	6022.76	5959.38
L_4					4161.41	4865.52	5041.13
L_5						3854.16	4243.19
L_6							3740.34

Table 3.32: The average time in milliseconds required to find the transition function resulting from the *most static information gain first* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	3953.91	2363.08	5702.82	10406.28	7036.28	2704.15	1704.96
L_1		975.99	2813.39	5574.47	4048.7	1753.71	1149.67
L_2			1573.21	3192.02	2676.73	1129.24	840.23
L_3				1709.03	1703.28	695.73	556.59
L_4					856.52	478.43	348.82
L_5						309.36	263.86
L_6							249.47

Table 3.33: The average of neighborhood sizes resulting from the *linear* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	31.40	17.71	13.16	11.55	8.79	7.15	5.56
L_1		21.49	14.13	12.00	8.91	7.16	5.61
L_2			20.42	13.52	9.45	7.41	5.86
L_3				20.44	11.26	8.01	6.11
L_4					17.41	9.73	6.84
L_5						15.43	8.57
L_6							14.97

Table 3.34: The average of number of rules in the transition function resulting from the *linear* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	172.97	208.73	233.66	233.24	235.82	247.54	248.03
L_1		208.29	232.87	232.93	235.89	247.47	248.05
L_2			233.53	232.32	235.78	247.48	248.25
L_3				230.6	235.46	247.72	248.1
L_4					234.92	247.8	248.31
L_5						247.31	248.21
L_6							247.35

linear order and present the result of using this order in the Tables 3.33 to 3.36

3.3.2.8 Removing Row by Row, Bottom Right First

To verify the effects of locality on the order of removing unimportant neighbors, we setup another experiments similar to of Section 3.3.2.7 but in a reverse order; i.e. the neighbors are sorted row by row, from the bottom right neighbor to the top left neighbor in the neighborhood of Step 3 in the Algorithm 3.1. We call this the *reverse linear* method and

Table 3.35: The average number of bits required to store the transition function resulting from the *linear* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	5652.49	7468.38	9486.30	10423.22	10091.02	10557.38	9632.77
L_1		4464.27	6761.45	8121.84	8186.22	8814.38	8329.31
L_2			4837.40	6092.39	6510.42	7291.50	7256.29
L_3				4588.55	5186.38	5926.64	6051.47
L_4					4031.42	4809.13	5090.41
L_5						3814.28	4253.92
L_6							3702.34

Table 3.36: The average time in milliseconds required to find the transition function resulting from the *linear* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	1838.6	851.16	1804.25	2957.66	1689.46	515.64	265.5
L_1		497.23	1116.92	1642.02	1036.64	351.72	197.16
L_2			806.78	1177.46	803.01	279.89	165.27
L_3				835.16	630.24	218.58	131.07
L_4					473.23	190.58	109.3
L_5						199.28	113.23
L_6							155.64

Table 3.37: The average of neighborhood sizes resulting from the *reverse linear* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	29.53	17.63	12.76	11.34	8.98	7.08	5.44
L_1		21.45	13.98	11.77	9.14	7.11	5.49
L_2			20.60	13.42	9.58	7.25	5.72
L_3				20.79	11.28	7.91	6.09
L_4					17.55	9.61	6.81
L_5						15.60	8.67
L_6							15.09

presents its results in the Tables 3.37 to 3.40

3.3.2.9 Removing in Random Order

In order to obtain a better understanding of the effects of each of the previous removal orders, we setup an experiment where the neighbors are examined for their importance in a totally random order. This can be considered as the base of comparison for each of the orders mentioned in the Sections 3.3.2.1 to 3.3.2.8. We call this the *random* order, and

Table 3.38: The average of number of rules in the transition function resulting from the *reverse linear* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	170.39	207.81	236.21	234.92	235.55	247.37	248.28
L_1		208.1	235.1	234.56	235.6	247.37	248.27
L_2			235.46	234.26	235.32	247.29	248.32
L_3				232.62	235.03	247.21	248.26
L_4					234.27	247.47	248.3
L_5						247.11	248.21
L_6							247.65

Table 3.39: The average number of bits required to store the transition function resulting from the *reverse linear* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	5071.91	7418.11	9379.34	10433.03	10219.10	10428.33	9417.90
L_1		4470.21	6761.73	8100.40	8321.19	8731.37	8156.25
L_2			5041.69	6146.17	6536.58	7125.16	7079.61
L_3				4736.60	5148.77	5838.52	6036.34
L_4					4035.13	4738.33	5066.48
L_5						3850.69	4303.55
L_6							3736.80

Table 3.40: The average time in milliseconds required to find the transition function resulting from the *reverse linear* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	2926.49	947.91	2070.78	3233.85	1834.6	534.71	251.49
L_1		526.64	1275.79	1988.5	1249.36	383.53	192.79
L_2			1018.03	1473.73	1000.25	296.56	174.46
L_3				1148.47	753.17	212.65	127.63
L_4					563.22	204.75	97.07
L_5						179.8	109.03
L_6							145.14

present its results in the Tables 3.41 to 3.44.

3.3.3 Analysis of the results - Expanding Neighborhood

In the follwoing sections we use bar plots to visualize the data of the tables of the format 3.7. To have an over-all observation of all the 9 methods at once, in the Sections 3.3.3.1 to 3.3.3.4 we provide an $X-Y-Z$ plot that bundles together all the 9 corresponding tables. In such case, the arrangement of the bars is as as illustrated in Figure 3.11.

Table 3.41: The average of neighborhood sizes resulting from the *random* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	30.71	17.48	13.38	11.53	8.91	7.03	5.38
L_1		21.40	14.38	11.88	9.02	6.95	5.46
L_2			21.08	13.63	9.42	7.30	5.70
L_3				20.99	11.26	8.01	5.91
L_4					17.66	9.59	6.66
L_5						15.54	8.53
L_6							15.05

Table 3.42: The average of number of rules in the transition function resulting from the *random* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	179.06	212.83	240.32	236.21	237.34	248.77	248.62
L_1		210.88	239.68	236.21	237.39	248.26	248.77
L_2			239.93	235.11	237.25	248.42	248.91
L_3				233.11	237.03	248.41	248.63
L_4					236.02	248.65	248.51
L_5						248.49	248.86
L_6							247.99

Table 3.43: The average number of bits required to store the transition function resulting from the *random* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	5567.76	7486.53	10039.59	10653.31	10296.52	10456.13	9330.78
L_1		4490.45	7123.16	8212.29	8337.31	8580.59	8128.27
L_2			5222.49	6240.44	6543.47	7235.97	7074.18
L_3				4784.81	5221.44	5951.04	5869.44
L_4					4107.88	4760.31	4959.23
L_5						3859.92	4246.72
L_6							3731.48

Table 3.44: The average time in milliseconds required to find the transition function resulting from the *random* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	2471.05	984.04	1904.09	3028.96	1728.8	520.34	259.48
L_1		573.39	1285.18	1922.98	1174.03	377.59	221.14
L_2			981.67	1450.42	902.94	304.36	165.59
L_3				1101.23	770.78	236.44	136.35
L_4					544.76	203.75	108.55
L_5						195.95	118.32
L_6							152.71

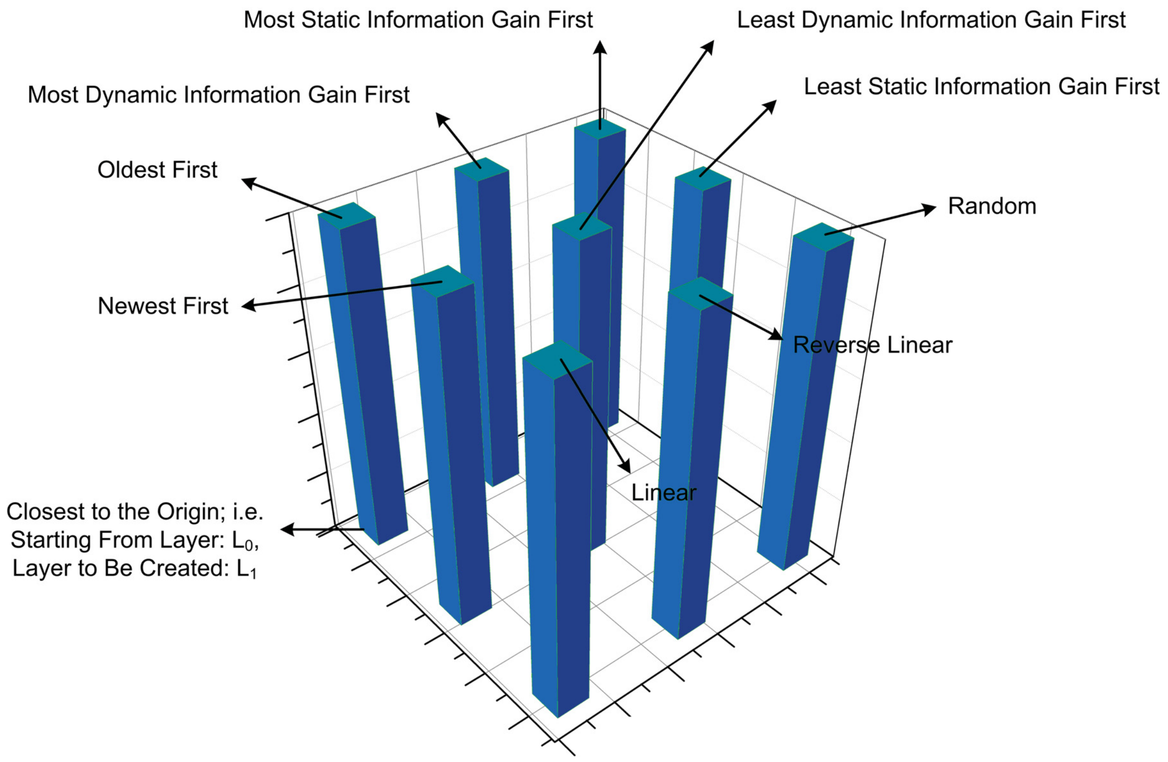


Figure 3.11: The arrangement of the bars where all the removing orders are bundled together in one plot.

3.3.3.1 Neighborhood Sizes

For better observation, Figure 3.12 illustrates different neighborhood sizes for different transition functions resulted form each of the 9 orders of evaluating neighbors to remove the unimportant ones that we explained in Sections 3.3.2.1 to 3.3.2.9. As it can be seen in that figure, for each transition function that creates a specific layer of the image from its previous layers, the result of all methods are bundled together to form groups of 9 bars adjacent to each other. The arrangement of bundles is explained in the Figure 3.11. Although contain different values, these bundles can be observed to have two certain trends, both expectable. First is that the Neighborhood Size grows larger as we keep the *layer to be created* constant and we start from earlier layers. For example creating Layer 3 of the images starting from the layer 0 (i.e. using the layers $0, 1 and 2$ in a CA with two steps of memory) needs obviously smaller neighborhood size comparing to creating the same layer 3 only from layer 2 of the image (i.e. a memoryless CA). This is because as the memory steps of the CA increase, CA has more information available to resolve the conflicts in the rule base. This means instead of growing the neighborhood larger to finally find a difference between two cells with different next states, CA can look in to the history of those two cells in a smaller neighborhood to find a difference and use it to distinguish the two.

The other characteristic of the plot in Figure 3.12 is that for a CA with a fixed levels of memory, the neighborhood size shrinks smaller as we move towards creating least significant layers. For example for a memoryless CA (i.e. memory steps = 0); creating the layer 2 from layer 1 of the image requires smaller neighborhood than the case of creating layer 1 form layer 0 of the image. This is again expectable because as we move towards less significant layers of the image, the cells with the same state become more spread in the image, providing more information to the algorithm to create the transition function from. This property (spreading through out the whole image in less significant bits) can be observed very well in the 8 figures in Table 3.1.

3.3.3.2 Number of Rules in the Transition Function

Figure 3.13 illustrates the average number of rules in each transition function resulted form each of the 9 orders explained in Sections 3.3.2.1 to 3.3.2.9. As it can be observed, the average number of rules is lower in the transition functions that create the more significant bits. This is because the cells with similar state are often group together in the more significant bits, resulting in more similarities and repeating patterns in lower levels (corresponding to more significant bits). This is in contrast to the less significant bits where cells with similar patterns are distributed all over the image. Again, this characteristics can be observed in the 8 figures of the Table 3.1.

The other property of the Figure 3.13 is that the number of rules in a transition function can easily get saturated to 250 rules. Remember that the size of the image is 20×25 pixels, meaning there are a total of 500 cells in CA, each corresponding to one pixel. This will result in an upper limit of 500 rules in the transition function. Knowing we store only the rules whose output are 1 and there are 128 images with no bias on the number of black or white pixels in each layer, it is reasonable to expect the upper limit of 250 rules in each transition function. On the other hand, this upper limit is quickly reached because the chance of having two exactly similar patterns of neighborhood in a CA with totally random

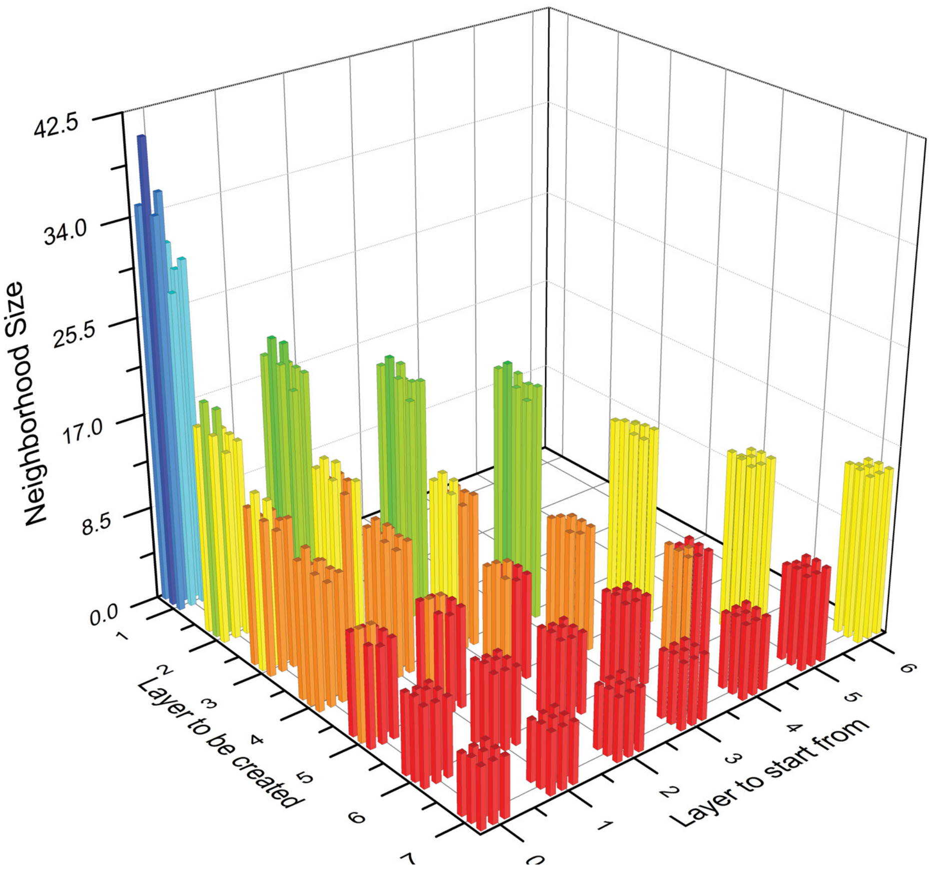


Figure 3.12: The overview of the neighborhood size for all transition functions resulting from different removing orders in the *Expanding Neighborhood* method.

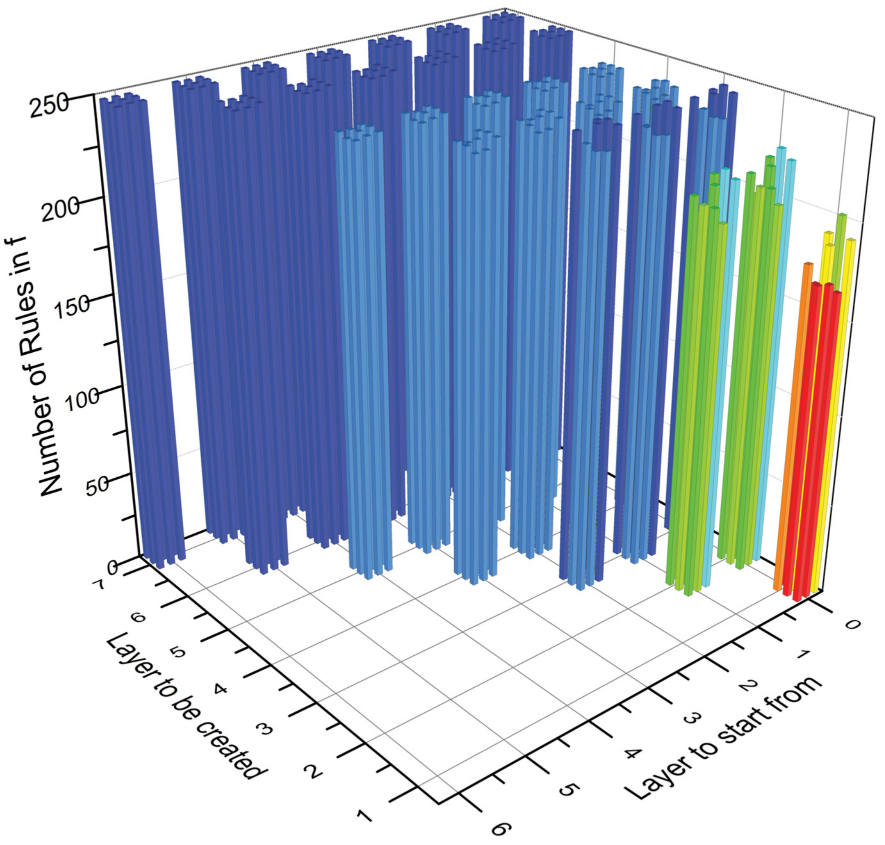


Figure 3.13: The overview of the number of rules in each transition functions resulting from different removing orders in the *Expanding Neighborhood* method.

configurations in time is $\frac{1}{(h+1)\times 2^n}$, where h is the number of memory steps of the CA and n is the neighborhood size. For an average neighborhood size of 20 for a memoryless CA this chance is $\frac{1}{1048576} = 0.00000095$, a very small chance. As the distribution of the cells with the same state becomes more random in less significant bits, less will be the chance of having two repeating rules in the transition function. Rules are merged together only if they are identical, so it is expectable to not merge any rules in creating higher levels of CA and hit the upper limit.

3.3.3.3 The Storage Size of the Transition Function

At the first glance one might expect to require less memory to store the transition functions with smaller neighborhood and expect the trend in the plot to be similar to of Figure 3.12, as storing the transition function is to store all the individual rules whose output is 1, and one bit per neighborhood in the *If* part of the rule will be required. Nevertheless, it is important to remember the effect of the number of memory steps of the CA on the rules. Although the neighborhood becomes smaller as we increase the number of memory steps of the CA, same neighborhood (with different configuration) appears multiple times in the *If* part of the rule. As it can be observer from the Figure 3.14, the effect of storing the same neighborhood multiple times cancels the effects of shrinking the neighborhood. In other terms, *adding history to the CA in the expanding neighborhood method does not have any positive effects on making the whole transition function smaller*.

The other trend that can be observed from the Figure 3.14 is that for a constant number of memory steps, the transition function for creating less significant layer takes less amount of memory. This is again consistent with our expectations of having more data available to the transition function in order to use smaller neighborhoods in case of less significant layers, similar to the same behavior in the neighborhood sizes.

3.3.3.4 The Calculation Time

Since the 3 first steps of the Algorithm 3.1 is common for all the 9 methods explained in Sections 3.3.2.1 to 3.3.2.9, we measure the time only for the forth step of the algorithm so we can compare the methods better. Figure 3.15 illustrates this measured time in milliseconds for finding each of transition functions. The first thing that is noticed in there is that two specific removing orders take extremely longer than the other orders. Not surprisingly, these two orders are *least dynamic information gain first* and *most dynamic information gain first*. One can easily tell that updating the information gain of all of the neighborhood elements after removing any of the unimportant elements is causing this extremely long time. One might rightfully believe that this extremely long time makes that specific order not practical even if its storage size is slightly better than other methods to store the resulted transition function.

To observe the overall characteristics of the calculation time we remove those two specific method from the plot of the Figure 3.15 and re-present the plot without those two orders in Figure 3.16.

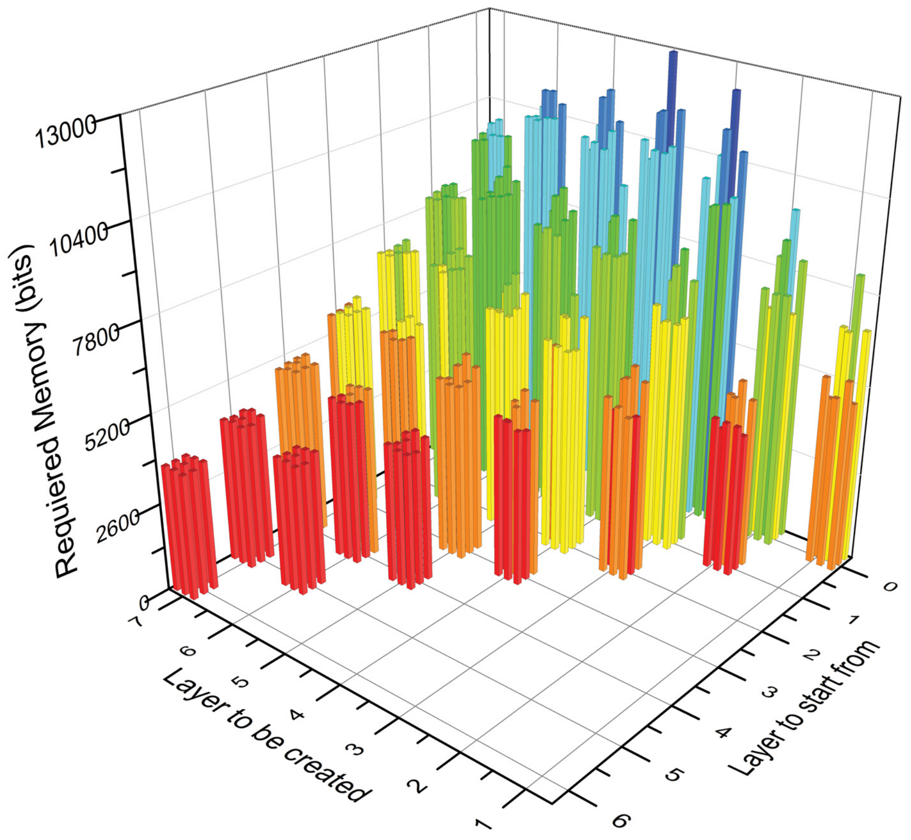


Figure 3.14: The storage size (in bits) required to store each of the transition functions resulting from different removing orders in the *Expanding Neighborhood* method.

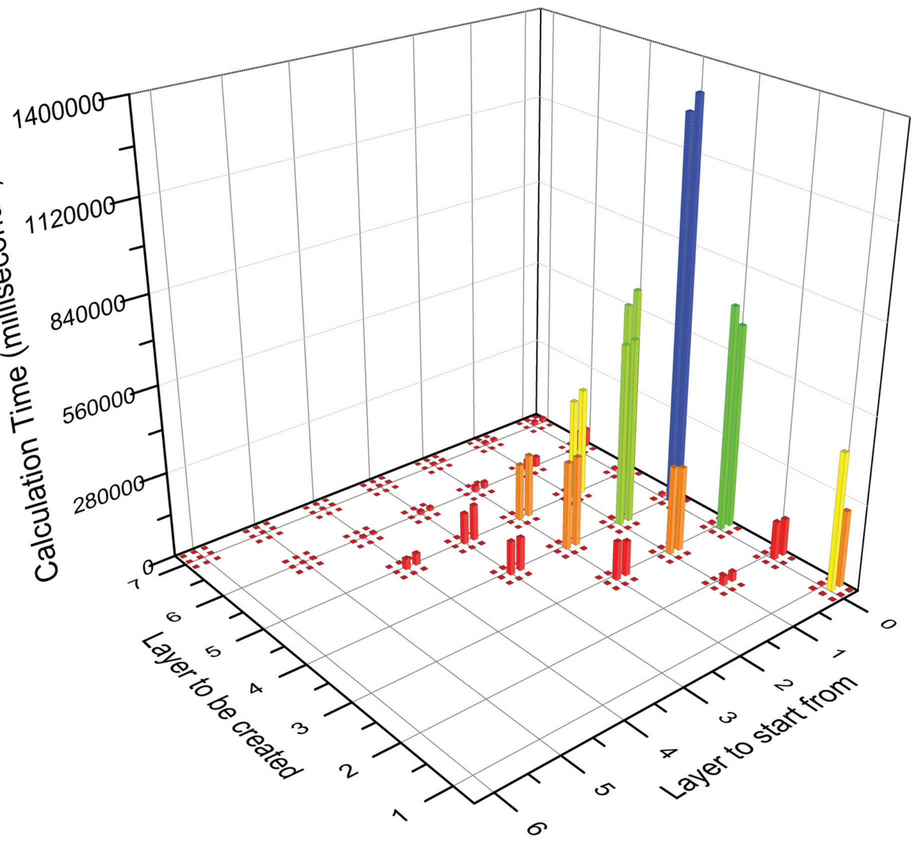


Figure 3.15: The time required for Step 4 of the Algorithm 3.1 to find the transition functions using each of the removing orders in the *Expanding Neighborhood* method.

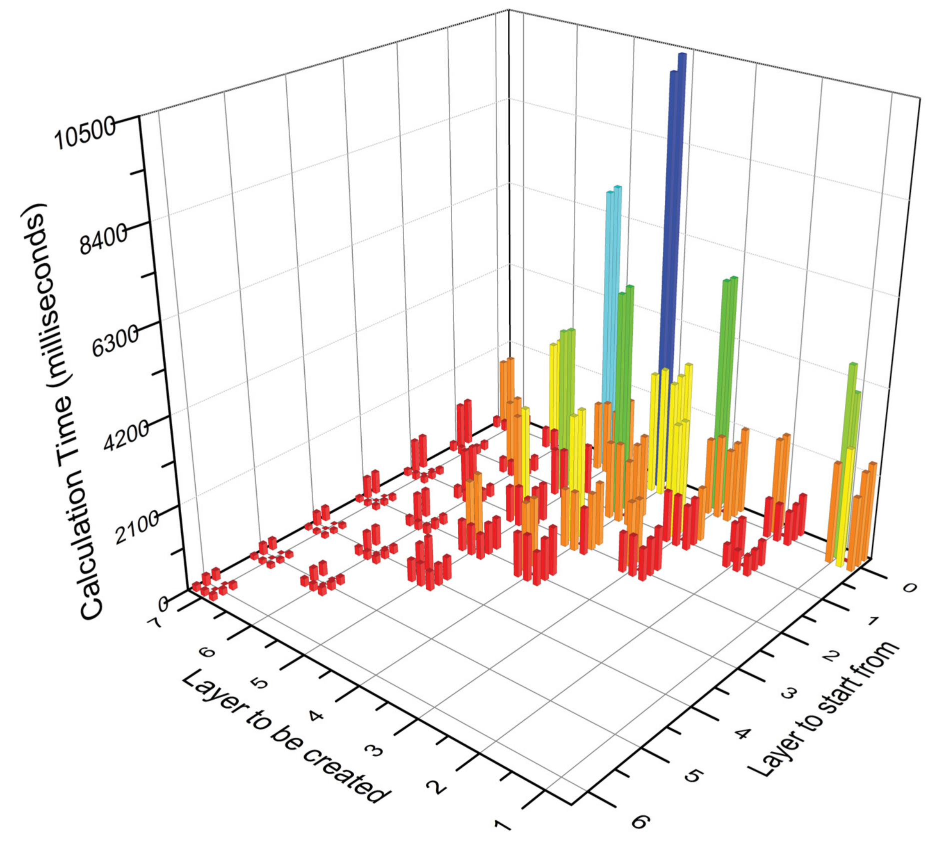


Figure 3.16: The time required for Step 4 of the Algorithm 3.1 to find the transition functions using each of the removing orders in the *Expanding Neighborhood* methods, except the *least dynamic information gain first* and *most dynamic information gain first*.

It is expected for the pattern of calculation time to follow the pattern of the storage sizes in the Figure 3.14 or in other terms, the pattern of total elements in the neighborhood when considering the memory of the CA. The reason is that the more elements in the *If* part of the rules are, the longer it takes to remove the unimportant ones and re-form the rules. This can be observed in the pattern of Figure 3.16. However, we emphasize again that the calculation time - when measured in seconds - is not an important feature of the algorithm. All the methods in the Sections 3.3.2.1 to 3.3.2.9 (except the *least information gain first* and *most information gain first* orders) have the same complexity and the differences in times are mainly because of implementation of the code. When comparing the time cost of the algorithms it is a much better idea to refer to the complexity analysis of each algorithm. In Section 3.2 such analysis is provided after introducing each algorithm.

3.3.3.5 The Effects of Removing Unimportant Neighbors According to Their Age

We define the age of an element in neighborhood the number of iterations of step 3 of the Algorithm 3.1 after the element was added to the neighborhood. Current existing methods sort the elements in the neighborhood according to their age and start removing the unimportant neighbors from the oldest one in neighborhood. To observe the effect of sorting by age there, we set up two experiments in Sections 3.3.2.1 and 3.3.2.2, and compare their results to the case where the neighbors are put in a random order in the beginning of Step 4 of the Algorithm 3.1 (i.e. the results of the Section 3.3.2.9). Figures 3.17 illustrates this comparison for the storage size of the resulting transition function. We compare only the storage size of the transition functions, because it is the key important property of a transition function as expressed in the Section 3.3.2. However, one can easily make the comparison for any of the other measured properties (i.e. neighborhood size, number of rules, calculation time) according to the provided tables in corresponding section. Note that in each plot in the following figures we compare 3 different orders at the same time. As it can be observed in the plots, there are bundles of 3 bars for each transition function from any starting layer to any target layer. The caption of the figures will explicitly mention what removing order each of the right, middle and the left bars are representing.

To our surprise, Figure 3.17 shows that removing the unimportant neighbors starting

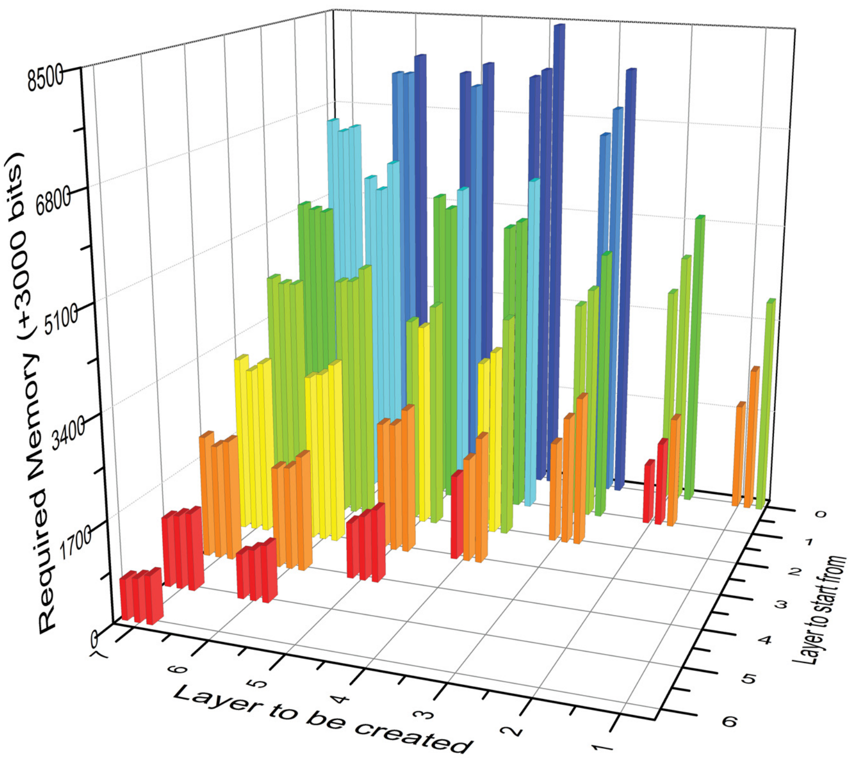


Figure 3.17: The comparison of the storage sizes the transition functions after using Remove Oldest First (right), Random (middle) and Remove Newest First (left) orders.

from the newest neighbor results in a smaller transition function. Moreover, the current method has a poorer performance even comparing to a random order. The reason is believed to be that our method adds groups of neighbors in each iteration of Step 3 of the algorithm 3.1, instead of adding them one by one. This was explained and justified in Section 3.2.1.2. As a result, there are chances that in the last iterations (that resolves all the conflicts) several extra (hence unimportant) neighborhood elements are added. These extra elements are the youngest elements and therefore examining the youngest elements first can be more effective. If started from the oldest ones, several of such originally unimportant neighbors can become important by the time they are examined, because one or more older, initially important neighbors might have been marked as not-important after adding the latest groups of elements.

3.3.3.6 The Effects of Removing Unimportant Neighbors According to Their Information Gain and Updating Their Information Gain After Each Removal

We told in Section 3.2.1.2 that the information gain of the elements in the *If* part of the rules seem to be a good candidate to sort the neighborhood elements according to, before examining for importance. Sections 3.3.2.3 and 3.3.2.4 provide the results of sorting once from the element with the least information gain and another time from the element with most information gain. In this experiment we update the information gain of all remaining elements whenever an unimportant neighbor is removed. Figure 3.18 compares the result of the two with the resulted of a random order.

Figure 3.18 makes it clear that while - as expected - removing the element with least information gain has a clear advantage over the opposite order, both orders act poorly when compared to a random order. The reason is believed to be the updating the information gain of the elements after each element is removed, because removing an element rearranges the combination of the remaining neighbors, and the information gain at the run time might not reflect the original importance of the neighbor anymore. Moreover, we remember from Section 3.3.3.4 that both methods take extremely long time to find the transition function. Considering their poor performance and long runtime, the author of this thesis deeply regrets even trying these methods.

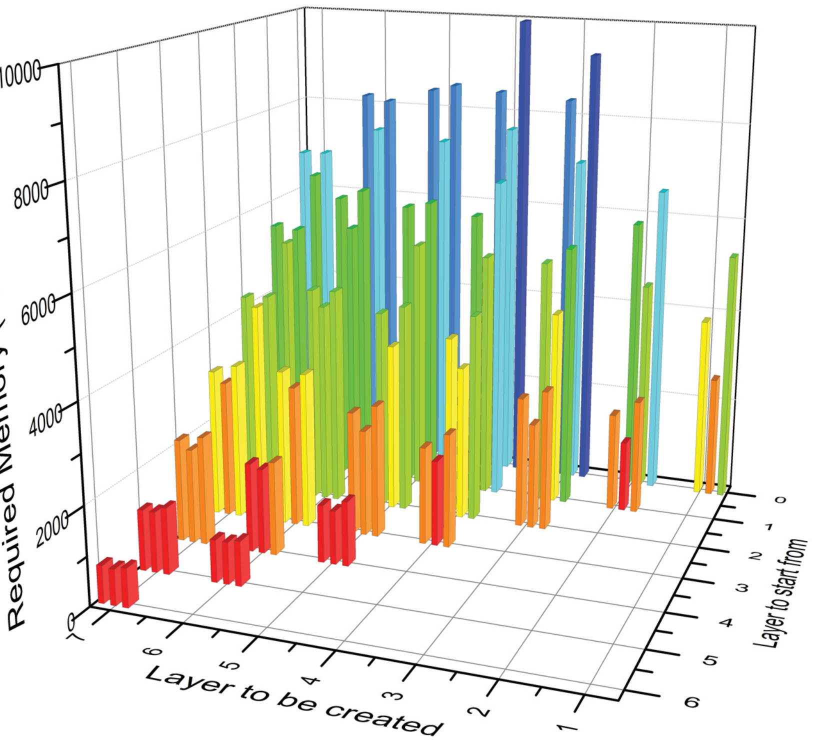


Figure 3.18: The comparison of the storage sizes of the transition functions after using Remove Most Dynamic Information Gain First (right), Random (middle) and Remove Least Dynamic Information Gain First (left) orders.

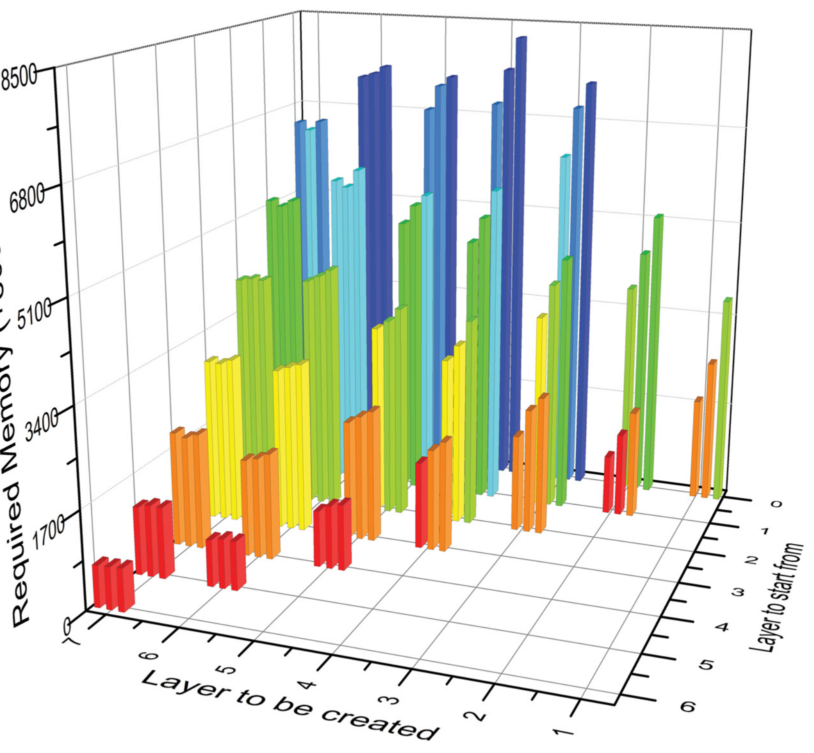


Figure 3.19: The comparison of the storage sizes of the transition functions after using Remove Most Static Information Gain First (right), Random (middle) and Remove Least Static Information Gain First (left) orders.

3.3.3.7 The Effects of Removing Unimportant Neighbors According to Their Information Gain Without Updating Their Information Gain After Each Removal

After observing the poor results of Sections 3.3.2.3 and 3.3.2.4 in Figure 3.18, we repeated the same experiment, this time without re-arranging the sorted list according to the updated information gains (Sections 3.3.2.5 and 3.3.2.6). This will let the elements to preserve their initial information gain, and also contributes largely to keep the run time relatively short. The results of those two experiments - as well as the random order as the base of comparison - is provided in Figure 3.19.

Figure 3.19 clearly proves the success of the idea of using information gain in sorting neighborhood elements for examining their importance (Section 3.2.1.2). As it can be seen in this figure, sorting the elements from the element with least initial information gain has a constant superiority over sorting in opposite order, and also almost always outperforms the random order. Remembering from the Section 3.3.3.3 that adding memory to CA does not contribute to smaller transition functions, we would want to create each layer only from its immediate previous layer. Therefor we can safely state that sorting from the element with the least initial information gain always out performs the random order in our application.

3.3.3.8 The Effects of Updating the Information Gains After Each Removal When Sorting By Information Gain

We already showed in the Sections 3.3.3.6 and 3.3.3.7 that updating the information gain of the neighborhood elements after removing each unimportant element has a negative effect on the storage sizes of the transition function. To demonstrate this effect more clearly we compare the two cases where in both the elements in the neighborhood are sorted from the element with least information gain to the one with the most information, but their information gain is constantly updated in one and does not get updated in the other. The results are illustrated in Figure 3.20. It is clear from that figure that changing the initial information gain of the elements has a negative impact on the storage sizes of the transition function.

3.3.3.9 The Effects of Removing Unimportant Neighbors According to Their Location in the CA

The second idea suggested in Section 3.2.1.2 was to examine the neighborhood elements according to their location in the CA. We did the experiments in the Sections 3.3.2.7 and 3.3.2.8. Figure 3.21 compares the cases where we sort the neighborhood elements in Step 4 of the Algorithm 3.1 row by row, once from the highest to the lowest row and once with the reverse order. We also provide the results of the random order for a better understanding of their performance.

As we expect, linear and reverse linear orders have similar performance, as they follow the same concept of locality; i.e. if a neighborhood element is unimportant, there are chanced that its adjacent element is unimportant too. Both method also have slightly better performance of the random order, proving that the idea or sorting based on the location

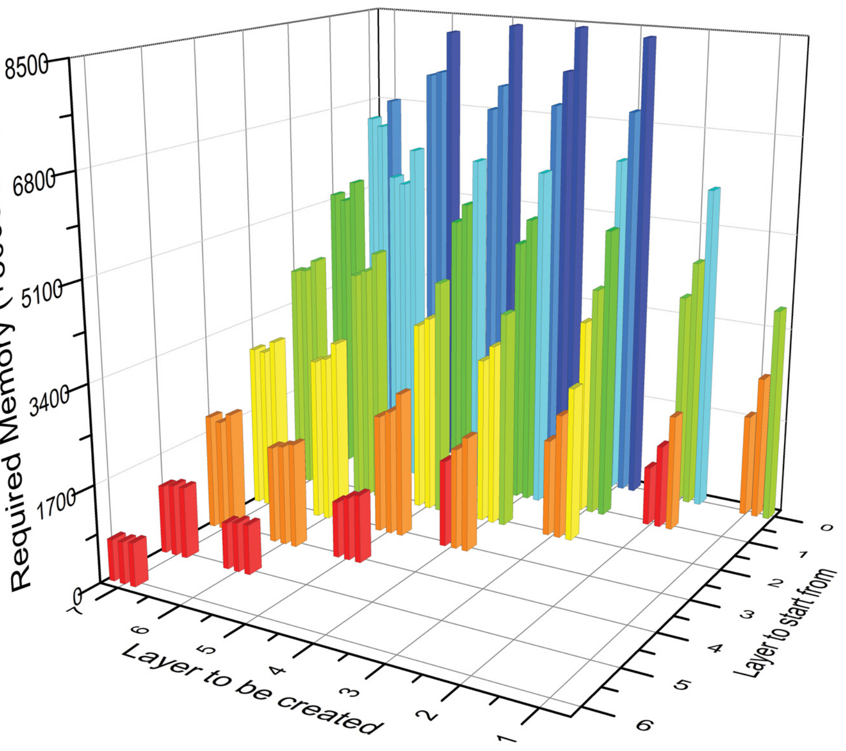


Figure 3.20: The comparison of storage sizes of the transition functions after using Remove Least Dynamic Information Gain First (right), Random (middle) and Remove Least Static Information Gain First (left) orders.

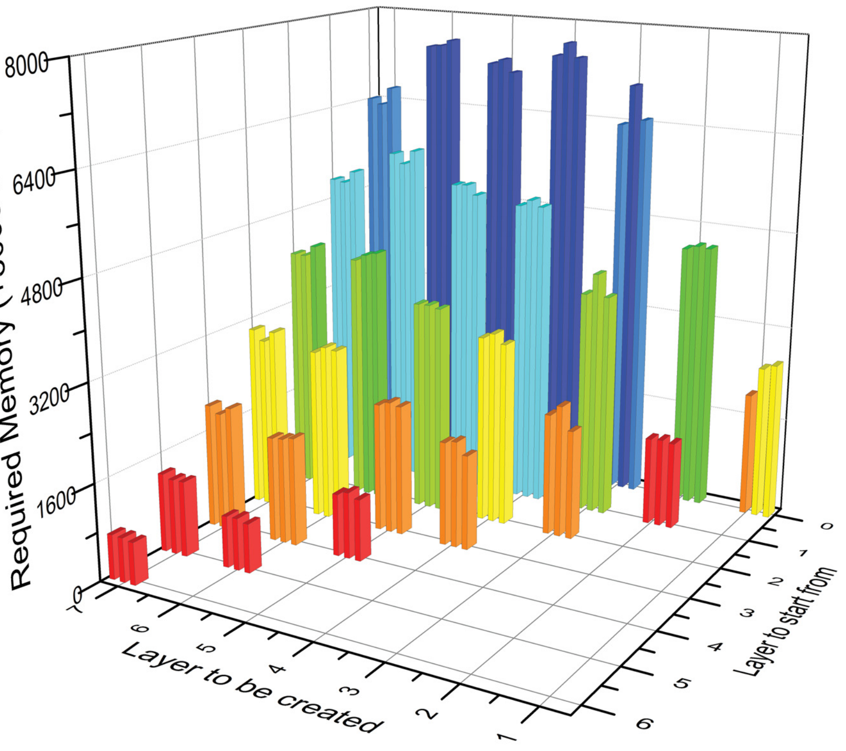


Figure 3.21: The comparison of the storage sizes of the transition functions after using the Linear (right), Random (middle) and Reverse Linear (left) orders.

of the elements suggested in Section 3.2.1.2 has been successful, although not providing a tremendous positive effect on the resulted transition function.

3.3.3.10 Comparison of the Most Successful Orders

We analyzed the effect of different properties (age, information gain, location) of the neighborhood elements in the above sections. We saw in Section 3.3.3.5 that it is better to examine newer neighborhood elements before the older ones. We saw in Section 3.3.3.7 and 3.3.3.8 that it is better to examine elements with less information gain before the ones with more information gain, and keep the initial information gain of the cells rather than dynamically updating the information gains. We also saw in the Section 3.3.3.9 that it is better to examine the physically close cells in a sequence rather than examining randomly located cells. Here we compare what we observed so far, to see using which of these parameters (age, information age and location) to sort the neighborhood elements in Step 4 of the Algorithm 3.1 results in the transition function with smallest storage size. Figure 3.22 compares the results of the best 3 orders so far.

As it can be observed in Figure 3.22, the order we suggested in Section 3.2.1.2; i.e. the middle bar which represents sorting from the neighborhood element with the least information gain; out performs the other orders in most of the times by finding a transition function whose storage requires the least number of bits in each cell of the CA.

3.3.3.11 Improvements to the Current Existing Methods

Table 3.45 represents the saving in the storage size of the transition function if our suggested *Least Static Information Gain First* order is used in Step 4 of the Algorithm 3.1 instead of the currently used *Oldest First* order. The average saving is 8.36% over all transition functions, and 12.65% for the transition functions of a memoryless CA (the diagonal values).

3.3.4 Results - Storing Individual Exceptions

To observe and compare the results of our first suggested method, i.e. using hidden states and storing individual exceptions, we implement the Algorithm 3.2 to run it on the same

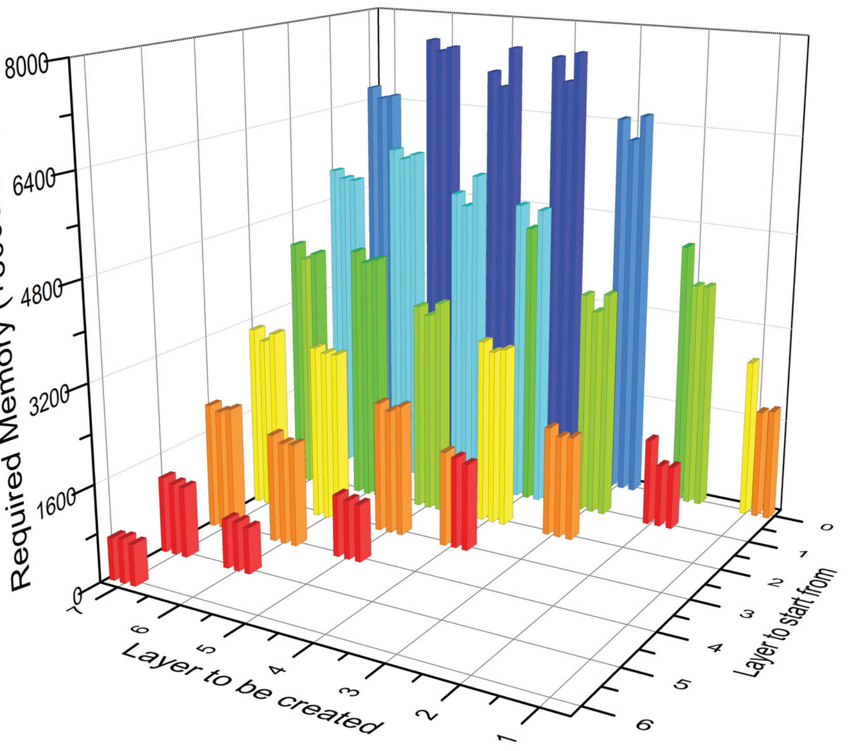


Figure 3.22: The comparison of the storage sizes of the transition functions after using the Oldest First (right), Least Static Information Gain First (middle) and Linear (left) orders.

Table 3.45: Saving in the storage size of the transition function if our suggested *Least Static Information Gain First* order is used in Step 4 of the Algorithm 3.1 instead of the currently used *Oldest First* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	29.40%	17.00%	14.73%	12.34%	7.98%	3.71%	-0.43%
L_1		18.36%	16.23%	13.23%	7.96%	4.25%	-1.43%
L_2			15.72%	12.94%	7.84%	4.59%	0.72%
L_3				12.53%	7.53%	4.43%	2.27%
L_4					6.26%	5.59%	1.07%
L_5						3.99%	2.91%
L_6							2.26%

Table 3.46: The average of number of rules in the transition function resulting from the Algorithm 3.2.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	50.07	122.05	179.69	199.94	218.53	238.51	247.78
L_1		99.63	177.58	199.76	218.53	238.51	247.78
L_2			139.74	198.79	218.51	238.5	247.78
L_3				156.32	217.96	238.5	247.78
L_4					172.57	237.96	247.71
L_5						187.13	247.44
L_6							193.59

set of inputs as described in the Section 3.3.1. Similar to the previous experiments, only the rules whose output are 1 are stored in the transition function. For this purpose we have made a minor modification to the Equation 3.29 in Algorithm 3.2, so that E is always equal to H_0. This lets us to always set $S_{out} = 1$ in step 3 of the algorithm. This will make a minor negative effect on the storage size of the transition function but at the same time it makes the representation of the transition function more similar to of Expanding Neighborhood method: similar to the Expanding Neighborhood method, the next state of a cell whose neighborhood cannot be found in the transition function will be 0. We will apply the same modification to the next suggested method as well, to make sure the representation is not favoring our suggested methods in the comparisons. We present the results in this section and provide the analysis of the results in the Section 3.3.5.

Unlike the existing method of Expanding Neighborhood, the Storing Individual Exceptions method has a fixed size of neighborhood. Tables 3.46 to 3.48 provide the average of number of rules in the transition function, storage size of the transition function (in bits) and the running time of algorithm to find the transition function in milliseconds.

3.3.5 Analysis of Results - Storing Individual Exceptions

3.3.5.1 Storage Size

Table 3.47 shows that in the *Storing Individual Exceptions* method, similar to the case of Expanding Neighborhood, adding memory to the CA does not improve the storage size of the transition function. We can see that to create any layer of the image, the smallest transition function is the one that build from the immediate previous layer only. In other

Table 3.47: The average number of bits required to store the transition function resulting from the Algorithm 3.2.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	1616.25	2909.27	5221.70	7808.84	10440.69	13241.25	15887.80
L_1		2049.06	3586.24	6006.56	8473.63	11094.61	13657.70
L_2			2257.66	4201.59	6506.67	8947.68	11427.59
L_3				2736.56	4537.81	6801.13	9197.49
L_4					2970.08	4650.16	6965.58
L_5						2922.66	4734.51
L_6							2965.70

Table 3.48: The average time in milliseconds required to find the transition function resulting from the Algorithm 3.2.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	15.8	15.68	16.44	16.11	17.85	19.11	25.1
L_1		15.72	15.5	15.52	15.66	16.93	19.09
L_2			15.78	15.61	15.51	15.54	17.69
L_3				15.7	15.56	15.6	15.71
L_4					15.6	15.64	15.62
L_5						16.09	15.59
L_6							15.61

words, the smallest values in each column of that table are those closest to the bottom. When compared to the storage size of the transition function resulting from the best case of *Expanding Neighborhood* method (Table 3.27), we notice that our first suggested method works considerably better for the CAs with no or one step of memory, as depicted in the Table 3.49. The improvement to the storage size of the transition function for the memoryless CA (the diagonal values) is 39.5%. When considered all the possible values (i.e. CAs with all number of memory steps), this method still improves the storage size by 3.31%.

3.3.5.2 Calculation Time

We analyzed the complexity of the Algorithm 3.2 in Section 3.2.2.3 and show that its order of complexity is lower than of the Algorithm 3.1 used for the *Expanding Neighborhood* method. This is no surprise then to notice the huge difference in the calculation time of the two methods. Table 3.50 illustrates the advantage of the calculation time in Table 3.48 over the calculation time in Table 3.28. It is clear there that the *Storing Individual Excep-*

Table 3.49: Savings in the storage size of the transition function when using the *Storing Individual Exceptions* instead of the best case of the *Expanding Neighborhood* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	66.51%	57.38%	42.93%	22.14%	-6.00%	-27.49%	-67.93%
L_1		49.43%	44.83%	22.56%	-5.96%	-27.80%	-66.27%
L_2			52.21%	29.39%	-1.84%	-25.51%	-62.53%
L_3				39.61%	10.89%	-15.87%	-56.29%
L_4					25.49%	1.10%	-38.86%
L_5						23.31%	-13.00%
L_6							20.40%

tions method has an absolute superiority in terms of calculation time over the *Expanding Neighborhood* method when using the Least Static Information Gain, as the former takes only 1.73%0 of the calculation time of the latter on average. Even when compared to the fastest order of *Expanding Neighborhood* (i.e. the Random order, where no time is spent on sorting the neighborhood elements), we can still observe the *Storing Individual Exceptions* method takes on average only 4.84% of the calculation time of *Expanding Neighborhood* method (Table 3.51).

Table 3.50: The ratio of the time needed to find the transition function in *Storing Individual Exceptions* method to the equivalent time in the *Expanding Neighborhood* method that uses the *Least Static Information Gain First* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	0.33%	0.66%	0.29%	0.16%	0.26%	0.71%	1.47%
L_1		1.60%	0.55%	0.28%	0.38%	0.97%	1.68%
L_2			0.97%	0.50%	0.60%	1.37%	2.14%
L_3				0.92%	0.95%	2.24%	2.94%
L_4					1.90%	3.38%	4.67%
L_5						4.93%	5.65%
L_6							5.89%

3.3.6 Results - Using Ranges of Values to Store Exceptions

We proposed the method of *Using Ranges of Values to Store Exceptions* in Section 3.2.3 and provided three algorithms to implement it (*Top-Down Search* in Section 3.2.3.2, *Bottom-Up Search* in Section 3.2.3.4 and *Evolutionary Search* in Section 3.2.3.6). Here we provide the

Table 3.51: The ratio of the time needed to find the transition function in *Storing Individual Exceptions* method to the equivalent time in the *Expanding Neighborhood* method that uses the *Random* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	0.64%	1.59%	0.86%	0.53%	1.03%	3.67%	9.67%
L_1		2.74%	1.21%	0.81%	1.33%	4.48%	8.64%
L_2			1.61%	1.08%	1.72%	5.11%	10.69%
L_3				1.43%	2.02%	6.60%	11.53%
L_4					2.86%	7.68%	14.39%
L_5						8.21%	13.18%
L_6							10.22%

Table 3.52: The average of number of rules in the transition function resulting from using the *Top-Down* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	34.1420	29	195.4	224.27	234.32	249.98	249.01
L_1		24	188.81	224.02	234.31	249.98	249.01
L_2			N/A	222.98	234.3	249.98	249.01
L_3				88.5	235.16	249.97	249.01
L_4					207	249.84	248.93
L_5						N/A	248.65
L_6							199

result of using each algorithm in the Sections 3.3.6.1 to 3.3.6.3, and compare and analyze the results in the Section 3.3.7. Similar to the method of Storing Individual Exceptions, this method has a fixed neighborhood size of 9, so same three parameters of Section 3.3.4 are measured for each algorithm here. These algorithms might fail because of the reason listed in Section 3.3.1, and in each experiment one table lists the number of times when this happens. Having a number n in row L_i and column L_j of that table means that the specific method has failed to find the transition function that generates the layer L_j of the image from the layers L_i to L_{j-1} in a total of n times (out of 128 times, the total number of images in the test data set).

3.3.6.1 Using the Top-Down search in the *Using Ranges of Values to Store Exceptions* method

Tables 3.52 to 3.55 provide the results of using Algorithm 3.4 in the *Using Ranges of Values to Store Exceptions* method that we defined in Section 3.2.3.

Table 3.53: The average number of bits required to store the transition function resulting from using the *Top-Down* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	458.57	617.66	5611.8	8349.08	10811.76	13757.81	15955.19
L_1		390	3759.31	6326.19	8702.63	11507.92	13714.03
L_2			N/A	4298.22	6593.72	9258.03	11472.87
L_3				1385	4509.83	7007.8	9231.71
L_4					3770	4763.82	6988.7
L_5						N/A	4746.88
L_6							3510

Table 3.54: The average time in milliseconds required to find the transition function resulting from using the *Top-Down* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	62925.9	123354.51	1268.94	92836.35	18243.41	1173.53	47.67
L_1		97820.18	1420.65	101553.07	19673.84	1094.38	39.83
L_2			5945.51	108866.35	20320.98	1098.19	35.05
L_3				76987.7	21592.38	1114.44	29.4
L_4					1722.26	1199.73	25.57
L_5						190.88	23.03
L_6							33.57

Table 3.55: the number of times using the *Top-Down* search in Section 3.2.3 exceeded the available resources.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	121	125	86	54	42	48	14
L_1		126	96	55	42	48	14
L_2			128	57	42	48	14
L_3				126	44	48	14
L_4					127	49	14
L_5						128	16
L_6							127

Table 3.56: The average of number of rules in the transition function resulting from using the *Bottom-Up* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	50.07	122.05	179.69	199.94	218.53	238.51	247.78
L_1		99.63	177.58	199.76	218.53	238.51	247.78
L_2			139.74	198.79	218.5	238.5	247.78
L_3				156.32	217.96	238.5	247.78
L_4					172.57	237.96	247.71
L_5						187.13	247.44
L_6							193.59

Table 3.57: The average number of bits required to store the transition function resulting from using the *Bottom-Up* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	1081.09	2882.63	5253.81	7577.03	10222.64	13222.73	15891.15
L_1		2082.89	3623.5	5775.31	8255.65	11076.09	13661.05
L_2			2583.04	3975.1	6288.93	8929.16	11430.95
L_3				2896.4	4323.9	6782.68	9200.85
L_4					3222.26	4635.63	6969.09
L_5						3450.7	4741.14
L_6							3565.00

3.3.6.2 Using the Bottom-Up search in the *Using Ranges of Values to Store Exceptions* method

Tables 3.56 to 3.59 provide the results of using Algorithm 3.5 in the *Using Ranges of Values to Store Exceptions* method (defined in Section 3.2.3).

3.3.6.3 Using the Evolutionary search in the *Using Ranges of Values to Store Exceptions* method

Tables 3.60 to 3.63 provide the results of using the Evolutionary Algorithm explained in the Section 3.2.3.6 in the *Using Ranges of Values to Store Exceptions* method. This method was defined in Section 3.2.3.

3.3.7 Analysis of Results - Using Ranges of Values to Store Exceptions

We again use the 3D bar plots to compare each property (number of rules, storage size and calculation time) of the transition function resulting from the *Using Ranges of Values to Store Exceptions* method. In each plot there will be three bars bundled together for each transition function. The rightmost bar represents the measurement of the property when

Table 3.58: The average time in milliseconds required to find the transition function resulting from using the *Bottom-Up* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	15.67	16.5	23.34	29.22	37.86	48.64	59.98
L_1		15.9	19.51	24.89	29.85	40.99	46.67
L_2			15.92	21.99	25.81	32.89	40.82
L_3				15.6782	23.2	27.96	34.75
L_4					15.57	25.06	29.64
L_5						15.94	24.39
L_6							15.86

Table 3.59: the number of times using the *Bottom-Up* search in Section 3.2.3 exceeded the available resources.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	0	0	0	0	0	0	0
L_1		0	0	0	0	0	0
L_2			0	0	0	0	0
L_3				0	0	0	0
L_4					0	0	0
L_5						0	0
L_6							0

Table 3.60: The average of number of rules in the transition function resulting from using the *Evolutionary* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	49.85	122.05	179.69	201.24	218.53	238.51	247.78
L_1		99.6326	177.58	199.76	218.53	238.51	247.78
L_2			139.74	199.44	218.5	238.5	247.78
L_3				156.51	217.96	238.5	247.78
L_4					172.57	237.96	247.71
L_5						187.13	247.44
L_6							193.59

Table 3.61: The average number of bits required to store the transition function resulting from using the *Evolutionary* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	1095.55	2881.69	5253.96	7623.24	10222.95	13221.64	15889.9
L_1		2079.14	3623.5	5778.75	8255.03	11075.15	13659.8
L_2			2582.26	3986.85	6287.84	8928.07	11429.7
L_3				2903.7	4323.28	6781.43	9199.6
L_4					3220.54	4634.53	6967.84
L_5						3448.2	4739.89
L_6							3564.53

Table 3.62: The average time in milliseconds required to find the transition function resulting from using the *Evolutionary* search in Section 3.2.3.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	5846.17	1768.84	621.07	1938.47	1464.61	326.21	117.02
L_1		3133.54	690.21	1806.84	1392.25	312.67	103.85
L_2			2289.65	1766.39	1349.35	306.58	99.08
L_3				3398.3	1490.11	305.39	92.96
L_4					3150.92	288.83	86.92
L_5						2096.29	86.55
L_6							2028.71

Table 3.63: the number of times using the *Evolutionary* search in Section 3.2.3 exceeded the available resources.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	2	0	0	2	0	0	0
L_1		0	0	0	0	0	0
L_2			0	1	0	0	0
L_3				1	0	0	0
L_4					0	0	0
L_5						0	0
L_6							0

Top-Down search (Section 3.2.3.2) is used. The middle bar represents the measurement of the same property when the *Bottom-Up* search (Section 3.2.3.4) is used and finally the leftmost bar represents the measurement of the same property when the *Evolutionary* search (Section 3.2.3.6) is used.

3.3.7.1 Number of Rules in the Transition Function

Figure 3.23 illustrates the average number of rules in the form of Equation 3.33 in the transition functions resulting from using any of the 3 search methods. We can see that the pattern of the number of rules is similar to of the case of *Extending Neighborhood* in Figure 3.13 but they reach to the upper limit of 250 rules more slowly. The reason is because of the capability of our second suggested method to merge multiple rules in to one rule. We can also notice a major difference between the results of *Top-Down* search and the other two in creating the first few layers in a memoryless CA. However, there is not enough data to strongly state the superiority of that method in those cases. As it can be seen in Table

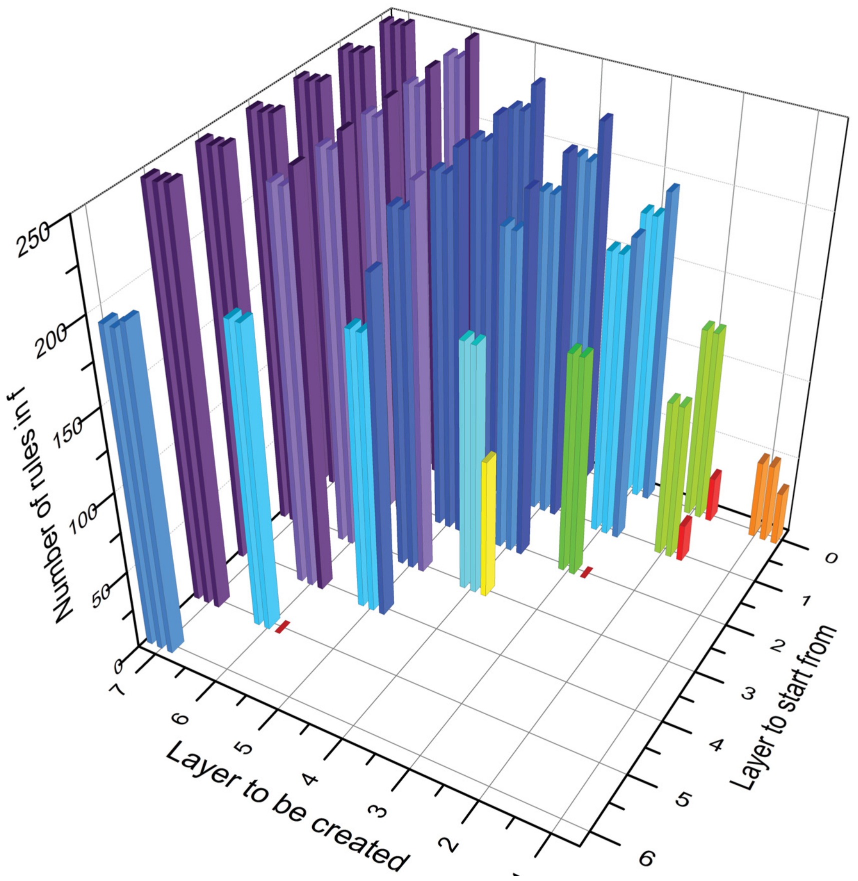


Figure 3.23: The overview of the number of rules in the transition functions in the *Using Ranges of Values to Store Exceptions* method using *Top-Down* (right), *Bottom-Up* (middle) and *evolutionary* (left) searches.

3.55, the reason is the failure of this method in finding the transition function for more than 90% of the input test data. As explained in Section 3.3.1, the *Top-Down* search method is forcefully terminated when the required runtime for finding one single transition function exceeds few hours.

3.3.7.2 The Storage Size of the Transition Function

Figure 3.24 provides an overview of the storage size of the transition functions resulting from *Using Ranges of Values to Store Exceptions* method and using any of the 3 search methods. A major difference between the trend of storage sizes here and in the *Expanding Neighborhood* method (Figure 3.14) is that in case of a memoryless CA (creating layer j from the layer $j-1$ only) the storage size shrinks as we move towards the higher layers

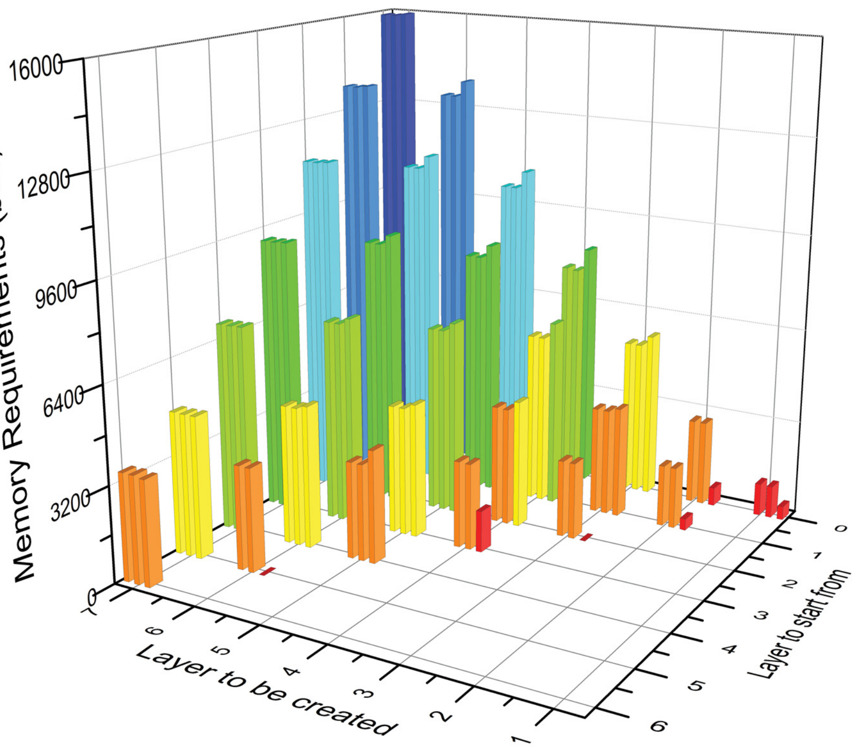


Figure 3.24: The overview of the storage size of the transition functions in the *Using Ranges of Values to Store Exceptions* method using *Top-Down* (right), *Bottom-Up* (middle) and *evolutionary* (left) searches.

(corresponding to the less significant bits) in the *Expanding Neighborhood* method, while the storage size is smaller in the lower layers in the *Using Ranges of Values to Store Exceptions* method. It is because the random distribution of the cells with the same state in higher layers contributes to provide more information to the Algorithm 3.1 and consequently reduces the neighborhood size and the storage size. On the other side, the same behavior (more random distribution) makes it difficult for any of the search methods in *Using Ranges of Values to Store Exceptions* to find larger *irregular regions* in the lower layers, so it is expected to have larger storage sizes there.

Once again, the results of the *Top-Down* search is not very dependable because of very few cases where it has been successful in finding the transition function.

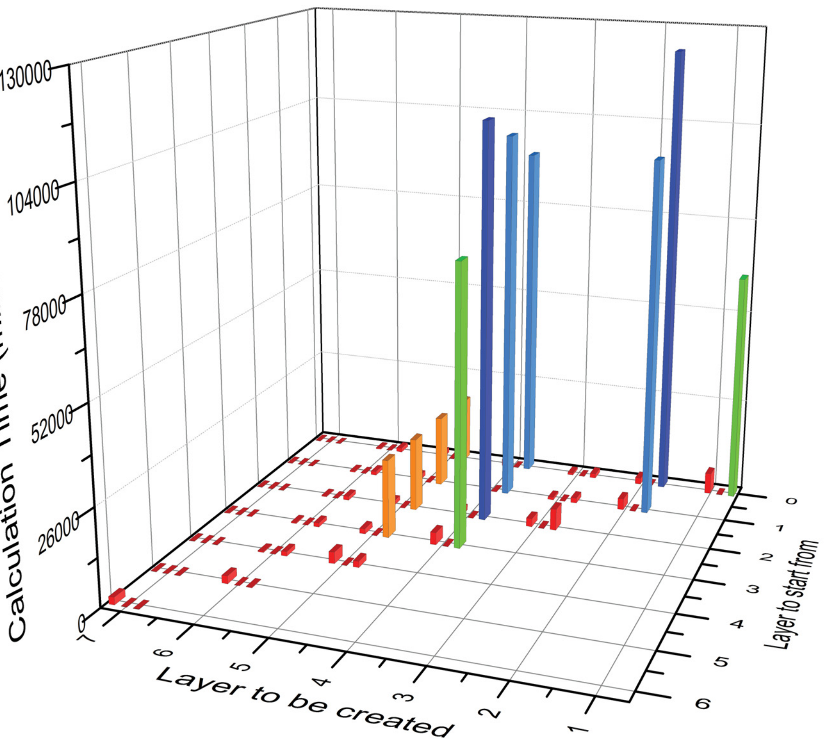


Figure 3.25: The overview of the calculation time of the transition functions in the *Using Ranges of Values to Store Exceptions* method using *Top-Down* (right), *Bottom-Up* (middle) and *evolutionary* (left) searches.

3.3.7.3 The Calculation Time for Each Search Method

Figure 3.25 compares the calculation time of the three different searches in the *Using Ranges of Values to Store Exceptions* method. The first thing that can be noticed in that plot is that the calculation time with the *Top-Down* search is so long that the difference between the other two cannot be observed. Remembering its poor success in finding the transition function in acceptable time we discard that method and re-draw the calculation time for the case of *Bottom-Up* and *Evolutionary* searches only in Figure 3.26.

As it can be seen in Figure 3.26, *Using Ranges of Values to Store Exceptions* method can find the transition function much faster when using the *Bottom-Up* search rather than the *Evolutionary* search method. Considering their very similar performance in storage size of

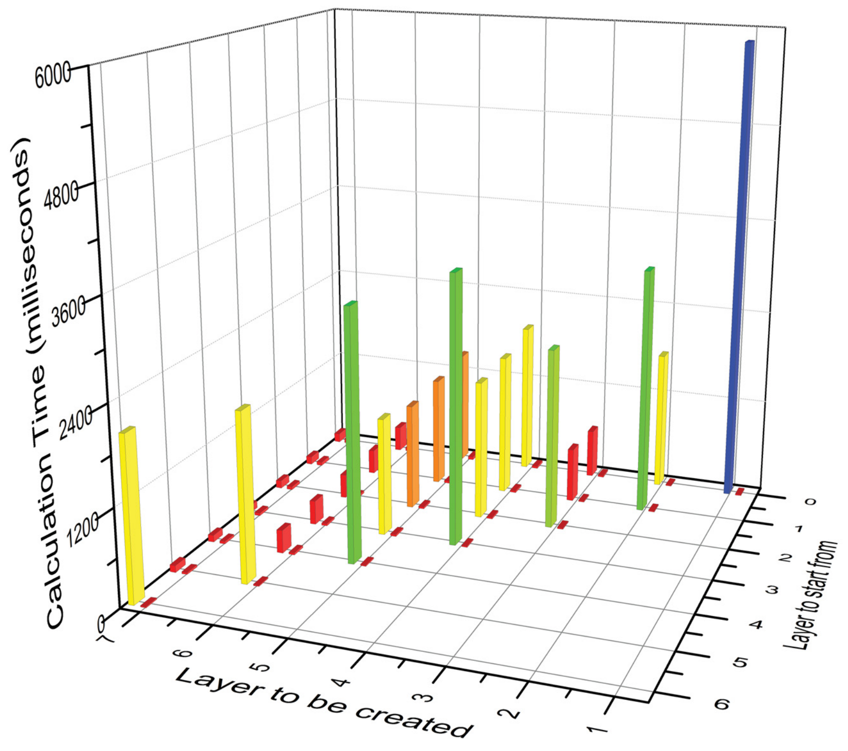


Figure 3.26: The overview of the calculation time of the transition functions in the *Using Ranges of Values to Store Exceptions* method using *Bottom-Up* (right) and *evolutionary* (left) searches only.

the transition function, we decide to use the *Bottom-Up* as the dominant search algorithm in *Using Ranges of Values to Store Exceptions* method when comparing it to the other methods of forming the transition function. The other reason to support this is the failure of the Evolutionary Search in finding the transition function in 6 out of 128 cases (Table 3.63).

3.3.8 Comparison of the Results

When comparing the performance of the two suggested methods in the storage size of the found transition function, we can notice a very small advantage for the *Storing Individual Exceptions* method over the *Using Ranges of Values to Store Exceptions* method. Table 3.64 shows that on average the storage size grows by 0.38% when using the latter instead of the former method. We do not compare the calculation time of the two methods since the average calculation time for each transition function has been few milliseconds (Tables 3.48 and 3.58). In that order the function used in our implementation (Microsoft WinAPI's function *GetTickCount()* on a non-real time operating system and a multi-thread environment) has a considerable error margin, making any comparison not dependable.

The improvement over the current existing method of *Expanding Neighborhood* is still impressive specially for the memoryless CAs (i.e. creating a layer from its immediately previous layer only). Table 3.65 shows an average improvement of 2.79% for CAs of all memory size, and 34.36% improvement for the memoryless CA. We need to remind our observation in previous methods that adding memory to the CA is not any beneficial in terms of storage size of the transition function, so the improvement of 34.36% will be the practical improvement. Tables 3.66 and 3.67 show that the *Using Ranges of Values to Store Exceptions* method takes only 2.74% and 8.72% of the time used by the best and fastest cases of *Expanding Neighborhood* method respectively.

3.3.9 A Note on Scalability

We mentioned in the end of Section 3.2.1 that the storage size of the found transition function using the current existing method of *Expanding Neighborhood* is not scalable. To show this and evaluate both our suggested methods, we ran all the experiments with the inputs

Table 3.64: Improvement of *Using Ranges of Values to Store Exceptions* method over the *Storing Individual Exceptions* method in Storage Size of the transition functions.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	33.11%	0.92%	-0.61%	2.97%	2.09%	0.14%	-0.02%
L_1		-1.65%	-1.04%	3.85%	2.57%	0.17%	-0.02%
L_2			-14.41%	5.39%	3.35%	0.21%	-0.03%
L_3				-5.84%	4.71%	0.27%	-0.04%
L_4					-8.49%	0.31%	-0.05%
L_5						-18.07%	-0.14%
L_6							-20.21%

Table 3.65: Improvement of *Using Ranges of Values to Store Exceptions* method over the best case of *Expanding Neighborhood* method (using the *LeastStatic Information Gain First* in Storage Size of the transition functions.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	77.60%	57.77%	42.57%	24.45%	-3.79%	-27.32%	-67.97%
L_1		48.59%	44.25%	25.54%	-3.24%	-27.58%	-66.31%
L_2			45.32%	33.20%	1.57%	-25.25%	-62.58%
L_3				36.08%	15.09%	-15.55%	-56.35%
L_4					19.16%	1.41%	-38.93%
L_5						9.46%	-13.15%
L_6							4.32%

Table 3.66: The ratio of the time needed to find the transition function in *Using Ranges of Values to Store Exceptions* method to the equivalent time in the *Expanding Neighborhood* method that uses the *Least Static Information Gain First* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	0.33%	0.70%	0.41%	0.29%	0.54%	1.81%	3.51%
L_1		1.62%	0.69%	0.45%	0.73%	2.34%	4.12%
L_2			0.98%	0.70%	1.00%	2.90%	4.92%
L_3				0.92%	1.42%	4.00%	6.51%
L_4					1.90%	5.42%	8.85%
L_5						4.88%	8.84%
L_6							5.98%

Table 3.67: The ratio of the time needed to find the transition function in *Using Ranges of Values to Store Exceptions* method to the equivalent time in the *Expanding Neighborhood* method that uses the *Least Static Information Gain First* order.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	0.63%	1.68%	1.23%	0.96%	2.19%	9.35%	23.12%
L_1		2.77%	1.52%	1.29%	2.54%	10.86%	21.11%
L_2			1.62%	1.52%	2.86%	10.81%	24.65%
L_3				1.42%	3.01%	11.83%	25.48%
L_4					2.86%	12.30%	27.30%
L_5						8.14%	20.62%
L_6							10.39%

Table 3.68: The average storage size of transition functions in a 40×50 CA resulting from using the *Oldest First* order in the *Expanding Neighborhood* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	54614.98	63365.92	85912.66	89812.25	89538.20	81458.53	68281.94
L_1		35236.80	59259.48	68899.05	71863.53	68235.27	58668.38
L_2			37459.39	49620.53	55335.59	55022.31	49439.26
L_3				33833.64	39836.56	42422.74	40620.75
L_4					27079.55	31704.86	32428.38
L_5						22516.27	25611.55
L_6							20020.22

four times larger than the original 20×25 images and follow the exact same experiment setup as in Section 3.3.1. Appendix A.2 contains the images used in this step.

3.3.9.1 Growths of the Storage Size of Transition Functions

Tables 3.68 lists the storage sizes of the transition functions for the new inputs when the current existing *Oldest First* order is used in the *Expanding Neighborhood* method. Tables 3.69 and 3.70 provide the same information for our improvements to this method, i.e. using the *Linear* and *Least Static Information Gain First* orders respectively. Tables 3.71 to 3.73 show the growth of the storage sizes when compared to the original experiments of Section 3.3.4.

When trying the first suggested method, i.e. Storing Individual Exceptions, we get the results as shown in Table 3.74. The growth of the storage sizes in regards to the original Table 3.47 is presented in Tabel 3.75.

Table 3.69: The average storage size of transition functions in a 40×50 CA resulting from using the *Linear* order in the *Expanding Neighborhood* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	51430.19	55539.89	79889.13	82048.03	84373.44	82113.33	69686.20
L_1		31295.88	55466.11	62814.52	67862.59	68578.55	59870.91
L_2			32866.35	44349.19	51689.34	55295.84	50285.94
L_3				29707.57	37079.06	42770.37	41007.31
L_4					25413.26	31721.41	33055.48
L_5						22598.18	25787.70
L_6							20015.74

Table 3.70: The average storage size of transition functions in a 40×50 CA resulting from using the *Least Static Information Gain First* order in the *Expanding Neighborhood* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	39149.35	51081.20	72295.88	75995.28	84904.34	77680.92	66988.36
L_1		28552.16	49338.39	58212.42	68309.72	64895.86	57464.81
L_2			30759.23	41426.84	51695.65	52705.97	48355.04
L_3				28231.62	36966.36	40591.22	39496.53
L_4					25506.95	30375.44	31825.69
L_5						21784.41	24893.69
L_6							19653.94

Table 3.71: The growth rate in the average storage size of transition functions resulting from using the *Oldest First* order in the *Expanding Neighborhood* method.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	7.99	7.71	8.01	7.85	8.37	7.55	7.25
L_1		7.10	7.64	7.71	8.27	7.53	7.24
L_2			6.68	7.26	7.98	7.36	6.98
L_3				6.53	7.23	6.91	6.75
L_4					6.37	6.37	6.40
L_5						5.67	5.93
L_6							5.25

Table 3.72: The growth rate in the average storage size of transition functions resulting from using the *Linear* order in the *Expanding Neighborhood* method.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	9.10	7.44	8.42	7.87	8.36	7.78	7.23
L_1		7.01	8.20	7.73	8.29	7.78	7.19
L_2			6.79	7.28	7.94	7.58	6.93
L_3				6.47	7.15	7.22	6.78
L_4					6.30	6.60	6.49
L_5						5.92	6.06
L_6							5.41

Table 3.73: The growth rate in the average storage size of transition functions resulting from using the *Least Static Information Gain First* order in the *Expanding Neighborhood* method.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	8.11	7.48	7.90	7.58	8.62	7.48	7.08
L_1		7.05	7.59	7.51	8.54	7.48	7.00
L_2			6.51	6.96	8.09	7.39	6.88
L_3				6.23	7.26	6.92	6.71
L_4					6.40	6.46	6.34
L_5						5.72	5.94
L_6							5.27

Table 3.74: The average storage size of transition functions in a 40×50 CA resulting from the *Storing Individual Exceptions* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	7087.33	9164.56	16379.72	29230.11	39285.98	49166.28	61557.59
L_1		9016.86	11833.42	22751.34	32091.48	41319.41	52976.80
L_2			10153.84	16276.43	24892.28	33471.76	44396.00
L_3				13139.31	17710.74	25622.91	35815.20
L_4					14274.13	17782.88	27226.91
L_5						14063.91	18639.88
L_6							14468.00

Table 3.75: The growth rate in the average storage size of transition functions resulting from the *Storing Individual Exceptions* method.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	4.39	3.15	3.14	3.74	3.76	3.71	3.87
L_1		4.40	3.30	3.79	3.79	3.72	3.88
L_2			4.50	3.87	3.83	3.74	3.88
L_3				4.80	3.90	3.77	3.89
L_4					4.81	3.82	3.91
L_5						4.81	3.94
L_6							4.88

Table 3.76: The average storage size of transition functions in a 40×50 CA resulting from using the *Bottom-Up* search in the *Using Ranges of Values to Store Exceptions* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	2424.58	6742.91	15922.31	27577.20	37634.48	48777.41	61607.94
L_1		6034.11	11276.55	21108.28	30442.79	40930.63	53027.14
L_2			9158.69	14694.77	23252.69	33083.35	44446.34
L_3				11915.22	16125.52	25235.91	35865.55
L_4					13531.16	17440.98	27277.91
L_5						14618.63	18748.34
L_6							15541.06

Table 3.77: The average storage size of transition functions in a 40×50 CA resulting from using the *Evolutionary* search in the *Using Ranges of Values to Store Exceptions* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	2778.41	6815.27	15912.44	27919.26	38153.53	48800.66	61607.38
L_1		6084.66	11275.61	21523.68	30803.00	40953.31	53025.08
L_2			9119.92	15135.00	23679.77	33102.48	44444.47
L_3				12248.31	16334.46	25255.59	35863.48
L_4					13718.86	17463.10	27276.59
L_5						14612.44	18746.84
L_6							15504.88

We try the same experiment using our second suggested method, *Using Ranges of Values to Store Exceptions* to compare the scalability of the storage size of the transition function. However, we do this when using the *Bottom-Up* and *Evolutionary* searches only, as we saw in Section 3.3.7 that the *Top-Down* search method is not successful in finding the transition function in most of the cases. Tables 3.76 and 3.77 provide the storage size of the transition functions in the new experiment setup for employing the *Bottom-Up* and *Evolutionary* searches respectively, and the Tables 3.78 and 3.79 show the growth of the transition function when compared to the case of 20×25 images; i.e. Tables 3.57 and 3.61.

3.3.9.2 Analysis of Results of Growths of the Storage Size of Transition Functions

The results of the Section 3.3.9.1 shows that our improvement to the existing method of *Expanding Neighborhood* has made it more scalable, as the *Least Static Information Gain* has the slowest growth rate among the variation of that method in Table 3.80. However, it

Table 3.78: The growth rate in the average storage size of transition functions resulting from using the *Bottom-Up* search in the *Using Ranges of Values to Store Exceptions* method is used.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	2.24	2.34	3.03	3.64	3.68	3.69	3.88
L_1		2.90	3.11	3.65	3.69	3.70	3.88
L_2			3.55	3.70	3.70	3.71	3.89
L_3				4.11	3.73	3.72	3.90
L_4					4.20	3.76	3.91
L_5						4.24	3.95
L_6							4.36

Table 3.79: The growth rate in the average storage size of transition functions resulting from using the *Evolutionary* search in the *Using Ranges of Values to Store Exceptions* method is used.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	2.54	2.37	3.03	3.66	3.73	3.69	3.88
L_1		2.93	3.11	3.72	3.73	3.70	3.88
L_2			3.53	3.80	3.77	3.71	3.89
L_3				4.22	3.78	3.72	3.90
L_4					4.26	3.77	3.91
L_5						4.24	3.96
L_6							4.35

is obvious from the Tables 3.68 to 3.73 that the growth in the size of transition functions resulting from the existing *Expanding Neighborhood* in general is faster than the growth of the size of the CA. Table 3.80 shows that the amount of memory required to store the transition functions resulting from the existing methods grows up about 7 times more when the size of the CA grows up by a factor of 4. The *Storing Individual Exceptions* reduces this growth rate to 3.96, very close to a linear ratio but the average storage size of the transition functions are still growing faster than the size of the CA for memoryless CAs (4.65). It is only the final method; i.e. *Using Ranges of Values to Store Exceptions* that has been successful to reduce this ratio to less than linear in both cases.

Table 3.80: The average growth of the storage size of the transition functions when the CA grows 4 times larger (4 times more cells in the CA)

Method	Variation	Average growth of the storage size for all memory sizes	Average growth of the storage size for memoryless CA
Expanding Neighborhood	Oldest First	7.14	6.51
Expanding Neighborhood	Linear	7.26	6.72
Expanding Neighborhood	Least Static Gain	7.09	6.47
Storing Individual Exceptions	N/A	3.96	4.65
Using Ranges of Values to Store Exceptions	Bottom-Up Search	3.64	3.66
Using Ranges of Values to Store Exceptions	Evolutionary Search	3.67	3.72

The reason why the storage size grows quickly in the current existing *Expanding Neighborhood* is that the chance of conflicts is more in larger CAs because of more cells. This means the neighborhood is expanded further, and consequently the chance of having two cells with the same neighborhood drops considerably. This means the method has to to store almost one rule for each cell in the rule base of the transition function. The number of rules is still close to (and therefore grows as fast as) the number of cells in the CA. On the other hand, we have the expanded neighborhood that contributes another coefficient to the total size of the rule base, making the growth faster than linear. This issue is addresses in both suggested methods. More specifically in the method of *Using Ranges of Values to Store Exceptions* the neighborhood size is kept constant no matter how far the CA grows. This means that there will be more chances that the same neighborhood repeats in more than one cell in the CA, hence reducing the rate of growth of the number of rules in the transition function (remember that we merge the rules with the same neighborhood in Step 2 of the Algorithm 3.2. This effect should be even more notable when the size of the CA grows by a larger factor.

3.3.9.3 Growths of the Calculation Time of Transition Functions

Similar to the storage size, the calculation time grows as the size of the CA grows. We provided the asymptotic analysis of each method in Section 3.2. Here we provide the practical calculation time of each method for the new CAs corresponding to the images in Appendix A.2 (4 times larger than the setup in 3.3.1) in Table 3.81 to Table 3.86. The growth rate of the calculation time can be obtained by comparing each table to its corresponding table for 20×25 images. This result is provided in the Tables 3.87 to 3.92. Note that the growth rate for the *Storing Individual Exceptions* method is not very trustworthy because of the limitations of the machine on which the code has been implemented. In practice, any measured time less than few milliseconds cannot be measured precisely on a non-real time operating system. This means some measured calculation time in the tables 3.48 and 3.84 are not very precise. Some growth rates in the Table3.90 as a result are marked with the question marks - meaning they are not precise results too.

3.3.9.4 Analysis of Results of Growths of the Calculation Time of Transition Functions

The summary of the growth in the calculation time is presented in Table 3.93. It can be seen that both improvements to the existing method; i.e. using *Linear* or *Least Static Information Gain First* instead of *Oldest First* order result in more scalable calculation time. Table 3.93 also confirms our initial claim in Section 3.2 that the complexity orders of the two

Table 3.81: The average calculation time of transition functions in a 40 × 50 CA resulting from using the *Oldest First* order in the *Expanding Neighborhood* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	308127.80	83525.06	344119.45	589396.51	272044.77	56158.78	10632.08
L_1		33748.66	170453.16	293717.41	150839.10	33279.34	7009.41
L_2			121849.67	208856.27	107400.86	22429.38	5262.54
L_3				153149.23	79731.74	15420.93	3926.80
L_4					47608.31	9684.46	2718.66
L_5						6576.18	3135.01
L_6							2978.13

Table 3.82: The average calculation time of transition functions in a 40 × 50 CA resulting from using the *Linear* order in the *Expanding Neighborhood* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	192216.18	62975.55	241030.06	443476.83	219254.20	56363.08	10971.85
L_1		24495.67	99091.99	186857.73	107482.38	32252.95	7193.24
L_2			66039.88	119793.39	72121.80	21653.24	5382.79
L_3				80693.17	51196.20	14504.20	4017.96
L_4					29634.95	9064.66	2823.16
L_5						6948.59	3230.47
L_6							3196.00

Table 3.83: The average calculation time of transition functions in a 40 × 50 CA resulting from using the *Least Static Information Gain* order in the *Expanding Neighborhood* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	509904.06	177226.84	859198.39	1665129.80	997556.16	256842.31	59757.19
L_1		50563.68	315719.10	700320.45	453416.52	135987.68	34473.41
L_2			162046.37	384510.31	266393.10	73566.08	22093.90
L_3				203684.75	151253.30	41878.44	13392.88
L_4					64900.41	20638.74	8001.47
L_5						10660.67	6982.61
L_6							4904.09

Table 3.84: The average calculation time of transition functions in a 40×50 CA resulting from using the *Storing Individual Exceptions* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	21.62	16.84	51.90	120.99	165.52	204.86	288.23
L_1		15.83	40.38	97.69	134.19	171.98	224.02
L_2			15.88	85.61	118.59	148.02	201.30
L_3				19.44	109.46	134.14	172.64
L_4					17.61	122.88	152.03
L_5						15.74	139.30
L_6							15.79

Table 3.85: The average calculation time of transition functions in a 40×50 CA resulting from using the *Bottom-Up* search in the *Using Ranges of Values to Store Exceptions* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	51.41	68.77	181.09	360.16	479.54	586.51	763.16
L_1		38.64	138.80	280.32	391.83	508.84	619.88
L_2			52.14	238.58	327.98	431.93	563.91
L_3				79.70	286.91	361.65	482.48
L_4					84.08	326.27	402.22
L_5						84.73	358.81
L_6							89.56

Table 3.86: The average calculation time of transition functions in a 40×50 CA resulting from using the *Evolutionary* search in the *Using Ranges of Values to Store Exceptions* method.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7
L_0	55469.81	24223.24	12037.52	19537.41	17152.73	6117.86	1985.73
L_1		38036.46	13585.08	19698.91	17001.49	6053.30	1908.63
L_2			27620.94	20830.02	17186.56	5954.77	1834.47
L_3				36037.78	17071.03	5905.20	1718.50
L_4					36405.73	5931.10	1694.03
L_5						28717.35	1799.70
L_6							26939.94

Table 3.87: The growth in the average calculation time of transition functions resulting from using the *Oldest First* order in the *Expanding Neighborhood* method, when the number of cells in the CA grows by a factor of 4.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	127.07	81.33	161.88	181.60	152.44	113.27	40.82
L_1		54.43	128.07	142.77	121.36	87.31	34.60
L_2			113.45	133.26	114.23	74.80	31.59
L_3				128.37	101.11	66.01	27.97
L_4					83.91	49.51	23.99
L_5						34.44	26.46
L_6							19.29

Table 3.88: The growth in the average calculation time of transition functions resulting from using the *Linear* order in the *Expanding Neighborhood* method, when the number of cells in the CA grows by a factor of 4.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	104.54	73.99	133.59	149.94	129.78	109.31	41.33
L_1		49.26	88.72	113.80	103.68	91.70	36.48
L_2			81.86	101.74	89.81	77.36	32.57
L_3				96.62	81.23	66.35	30.65
L_4					62.62	47.56	25.83
L_5						34.87	28.53
L_6							20.53

Table 3.89: The growth in the average calculation time of transition functions resulting from using the *Least Static Information Gain* order in the *Expanding Neighborhood* method, when the number of cells in the CA grows by a factor of 4.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	108.00	75.09	150.22	165.25	142.85	95.37	34.96
L_1		51.41	111.58	127.46	110.37	77.76	30.40
L_2			100.03	122.01	103.16	64.85	26.65
L_3				119.00	92.58	59.96	25.08
L_4					79.22	44.62	23.90
L_5						32.64	25.29
L_6							18.49

Table 3.90: The growth in the average calculation time of transition functions resulting from using the *Storing Individual Exceptions* method, when the number of cells in the CA grows by a factor of 4.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	1.37?	1.07?	3.16	7.51	9.27	10.72	11.48
L_1		1.01?	2.61	6.29	8.57	10.16	11.73
L_2			1.01?	5.48	7.64	9.52	11.37
L_3				1.24?	7.03	8.59	10.99
L_4					1.13?	7.85	9.73
L_5						0.98?	8.93
L_6							1.01?

Table 3.91: The growth in the average calculation time of transition functions resulting from using the *Bottom-Up* search in the *Using Ranges of Values to Store Exceptions* method, when the number of cells in the CA grows by a factor of 4.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	3.28	4.17	7.76	12.32	12.66	12.06	12.72
L_1		2.43	7.11	11.26	13.13	12.41	13.28
L_2			3.27	10.85	12.71	13.13	13.81
L_3				5.08	12.36	12.93	13.88
L_4					5.40	13.02	13.57
L_5						5.31	14.71
L_6							5.65

Table 3.92: The growth in the average calculation time of transition functions resulting from using the *Evolutionary* search in the *Using Ranges of Values to Store Exceptions* method, when the number of cells in the CA grows by a factor of 4.

	L_0	L_1	L_2	L_3	L_4	L_5	L_6
L_0	9.49	13.69	19.38	10.08	11.71	18.75	16.97
L_1		12.14	19.68	10.90	12.21	19.36	18.38
L_2			12.06	11.79	12.74	19.42	18.51
L_3				10.60	11.46	19.34	18.48
L_4					11.55	20.53	19.49
L_5						13.70	20.79
L_6							13.28

suggested methods in this thesis (*Storing Individual Exceptions* and *Using Ranges of Values to Store Exceptions*) are less than the complexity order of the current existing *Expanding Neighborhood* method. Adding this observation to the previous ones, that the storage sizes of transition functions are both smaller and more scalable in these two methods, we can conclude that for the inverse problem of the Cellular Automata, when the configuration of the CA is forced from an unknown external models and not necessarily from a developmental model such as the test data sets here, our two suggested methods in this thesis demonstrate a considerably better performance than the current existing method.

Table 3.93: The average growth of the calculation time of the transition functions when the CA grows 4 times larger (4 times more cells in the CA)

Method	Variation	Average growth of the calculation time for all memory sizes	Average growth of the calculation time for memoryless CA
Expanding Neighborhood	Oldest First	87.69	80.14
Expanding Neighborhood	Linear	75.15	64.33
Expanding Neighborhood	Least Static Gain	79.22	72.68
Storing Individual Exceptions	N/A	8.74	N/A
Using Ranges of Values to Store Exceptions	Bottom-Up Search	10.01	4.35
Using Ranges of Values to Store Exceptions	Evolutionary Search	15.23	11.83

3.4 Conclusion

In this chapter we showed how Cellular Automata can be used for pattern generation problems. We studied the inverse problem of Cellular Automata, where the consecutive configurations are known and the neighborhood and transition function are to be found.

We pushed our methods to their limits, by using consequent configurations which are not resulting from a developmental encoding. As explained in the text, the configurations in this chapter are copied from the distribution of the n$^{\text{th}}$ bit in a 256-level gray scale images of faces. While the distribution of the n^{th} and $(n+1)^{th}$ bits are not totally independent from each other in such images, knowing that the $(n+1)^{th}$ bit is not directly the result of applying a well-defined function on the n^{th} bit makes the problem more challenging in the inverse problem. We showed that the current existing methods can be improved if the suggested modifications are applied. We also suggested two new methods - with new formats of the transition function - that contribute to both calculation time and storage size of the resulted transition function. The latter parameter, i.e. the storage size of the resulted transition function, is believed to be the most important characteristic of the method when the solution is to be used on actual hardware. We also showed that our suggested methods are more scalable, since the size of the solution in our methods grows much slower than of the current methods.

The size of the solution on average grows more than 7 tiles larger when the pattern has grown 4 times larger. We could reduce this ratio to less than linear, yet not as low as one might expect from a developmental method. There is an important reason for this: not every pattern with any arbitrary configurations over time can be stored in a small developmental program and be regenerated using any arbitrary resolution. Remember that our transition functions are lossless. For the patterns that have not been the result of a developmental growth, there is a price for size and resolution of the generated pattern and this price is reflected in the size of the transition function. The natural development encoding is benefiting from an excellent scalability because it is not solving the inverse problem. Natural evolution was never given the final pattern to grow, neither was provided with the desired functionality. The phenotypes have been evolved because they have been successful in their environment, without trying to achieve any pre-defined functionality of architecture. Yet for an inverse problem with a non-developmental pattern configuration, our second suggested method is scalable since the its solutions grow slower than the size of the problem.

On the other hand, what is creating a negative effect on scalability is providing us robustness and fault tolerant: as the size of the exceptions or irregular regions grow in our methods, there will be less chances that an error in the configuration of the Cellular Automata gets propagated in the neighborhood. The reason is that now that the size of the details of the rules are increased, if an error is injected and a cell is forcefully moved to a faulty state, the new faulty neighborhoods have less chance to be found in the *if* part of any rule in the rule base. This means the forcefully modified cell can calculate its state back to the original by using the previous states of its neighborhood and hidden states, before causing an error in the adjacent cells.

The possible improvement for future researchers is to find better search methods for solving the *Minimum Irregular Regions Problem* for our second method defined in 3.1. However, better search algorithms might be specific to the characteristics of the images whose layers are forming the configuration of the CA.

Chapter 4

Summary

4.1 Generating Patterns When the Functionality is Known

In Chapter 2 we proposed a developmental approach to emerge patterns of unknown architecture to demonstrate given functionalities. We explained a method for evolving the developmental programs that describe gate level feed forward digital circuits with local connections. Keeping the connections local will eliminate the routing overhead and difficulties on the configurable circuits if ever decided to employ this method on actual programmable hardware.

We also introduced an innovative fitness function that uses the *sensitivity analysis* (presented in this thesis for the first time), and showed that the evolution is almost 4 times more successful in finding solutions for certain problems if it uses our fitness function instead of the traditional fitness function used in EHW.

In the end we showed that the solutions are scalable for modular circuits such as parity generators, since the algorithm can find the optimum solution of the given number of inputs faster if it already has the answer to the problem of smaller input size. This is an important achievement in Evolutionary Hardware Design, where the answer to each given problem used to be evolved from scratch, needing exponentially longer time after an increase in the problem size.

4.2 Generating Patterns When the Architecture is Known

Cellular Automata has been used for developing patterns with given topology and architecture. There has been methods for solving the inverse problem of Cellular Automata when the consecutive patterns are known. In Chapter 3 we introduced two separate improvements to the current existing method of solving the inverse problem of Cellular Automata. We showed that using the locality or information gain of the state of the cells in removing the unimportant neighbors contributes to smaller size for the transition function, which is a critical feature of the final results if the method is to be used in real hardware.

We also suggested two methods that improve both time and memory efficiency. We showed that adding hidden states to the Cellular Automata can result in shorter calculation time and solutions with smaller sizes. Moreover, we showed that our methods are considerably more scalable in terms of both storage size and calculation time of the solution. In our experiments they could reduce the growth rate of the transition function to less than the growth rate of the problem size. This is while the current existing methods in our experiments grew almost two times faster than the problem size. Also keeping the main neighborhood size constant and using cell's own hidden states in the transition function greatly reduces the calculation time for the transition function. Finally, when using this method on real programmable hardware we will require less resources spent on routing because the neighborhood includes only the adjacent cells. This will save more resources for calculations and data storage on such hardware.

4.3 Conclusion and Future Steps

In this thesis we showed that developmental descriptions are capable of emerging patterns who posses certain form or functionality. Such developmentally generated pattern inherits the built-in features of developmental systems such as scalability and fault tolerance, given the pattern has enough regularity and modularity. The methods suggested in this thesis improved the following measures:

- Improved Evolvability: Our fitness function in Section 2.2.3 increases the chance of evolution to find the solution.

- Improved Efficiency: Our improvements in Section 3.2.1.2 decrease the size of transition functions in the inverse problem of Cellular Automata.
- Reduced Complexity: Our methods in Sections 3.2.2 and 3.2.3 reduce the complexity of the algorithm to find the transition functions in the inverse problem of Cellular Automata.
- Improved Scalability: Our method in Sections 3.2.2 and 3.2.3, and more specifically Section 3.2.3.4 improve the scalability of the algorithm to find the transition functions in the inverse problem of Cellular Automata.

The research explained in this thesis has the potential to be extended and improve the formation of large patterns even further. Following items can be considered for the future steps of this topic for the researchers interested in this field.

- Finding hidden transition functions for different applications.
- Finding suitable patterns with enough regularity and modularity to be generated using development.
- Finding Better algorithms for solving the Minimum Irregular Regions Problem.
- Using evolution to find optimum neighborhood directly, in oppose to fixing them to a certain neighborhood.
- Forming different transition functions for different regions, in oppose to using the same transition function for the whole pattern.
- Shrinking neighborhood in the last methods, form a fixed size of nine neighbors to a variable size that minimizes the size of transition function.

Bibliography

[1] A. Adamatzky, *Automatic programming of cellular automata: identification approach*, Kybernetes **26** (1997), no. 2, 126–135.

[2] ———, *Game of life cellular automata*, Springer, 2010.

[3] R. Alonso-Sanz, *Cellular automata with memory*, Éd. des Archives contemporaines, 2008.

[4] P.J. Angeline, *Morphogenic evolutionary computations: Introduction, issues and examples*, Evolutionary Programming IV: The Fourth Annual Conference on Evolutionary Programming, 1995, pp. 387–401.

[5] P. Bentley, *Evolutionary design by computers*, vol. 1, Morgan Kaufmann, 1999.

[6] Peter Bentley and Sanjeev Kumar, *Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem*, Proceedings of the Genetic and Evolutionary Computation Conference, vol. 1, Morgan Kaufmann, 1999, pp. 35–43.

[7] P.J. Bentley, T.G.W. Gordon, J. Kim, and S. Kumar, *New trends in evolutionary computation*, Evolutionary Computation, 2001. Proceedings of the 2001 Congress on, vol. 1, IEEE, 2001, pp. 162–169.

[8] Elwyn R Berlekamp, John Horton Conway, and Richard K Guy, *Winning ways for your mathematical plays. volume 2*, AK Peters, 1982.

[9] J.L. Beuchat and J.O. Haenni, *Von neumann's 29-state cellular automaton: a hardware implementation*, Education, IEEE Transactions on **43** (2000), no. 3, 300–308.

[10] D. Bleh, T. Calarco, and S. Montangero, *Quantum game of life*, EPL (Europhysics Letters) **97** (2012), no. 2, 20012.

[11] R.D. Carr, S. Doddi, G. Konjevod, and M. Marathe, *On the red-blue set cover problem*, Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2000, pp. 345–353.

[12] Santanu Chattopadhyay, Shelly Adhikari, Sabyasachi Sengupta, and Mahua Pal, *Highly regular, modular, and cascadable design of cellular automata-based pattern classifier*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **8** (2000), no. 6, 724–735.

[13] F. Corno, M.S. Reorda, and G. Squillero, *Exploiting the selfish gene algorithm for evolving hardware cellular automata*, Evolutionary Computation, 2000. Proceedings of the 2000 Congress on, vol. 2, IEEE, 2000, pp. 1401–1406.

[14] A. Deutsch, S. Dormann, and P.K. Maini, *Cellular automaton modeling of biological pattern formation: characterization, applications, and analysis*, Springer, 2005.

[15] Asbjoern Djupdal and Pauline C Haddow, *Evolving redundant structures for reliable circuits-lessons learned*, Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on, IEEE, 2007, pp. 455–462.

[16] M. Ebne-Alian and N. Kharma, *A new method to find developmental descriptions for digital circuits*, Evolvable Systems: From Biology to Hardware (2010), 73–84.

[17] M. Elkin and D. Peleg, *The hardness of approximating spanner problems*, STACS 2000, Springer, 2000, pp. 370–381.

[18] Niloy Ganguly, Pradipta Maji, Sandip Dhar, Biplab Sikdar, and P Chaudhuri, *Evolving cellular automata as pattern classifier*, Cellular Automata (2002), 56–68.

[19] A.S. Georghiades, P.N. Belhumeur, and D.J. Kriegman, *From few to many: Illumination cone models for face recognition under variable lighting and pose*, IEEE Trans. Pattern Anal. Mach. Intelligence **23** (2001), no. 6, 643–660.

[20] Scott F. Gilbert, *Developmental biology*, 4th ed., Sinauer Associates Inc, April 1994.

[21] T.G.W. Gordon, *Exploring models of development for evolutionary circuit design*, Evolutionary Computation, 2003. CEC'03. The 2003 Congress on, vol. 3, IEEE, 2003, pp. 2050–2057.

[22] T.G.W. Gordon and P.J. Bentley, *Towards development in evolvable hardware*, Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on, IEEE, 2002, pp. 241–250.

[23] ______, *Bias and scalability in evolutionary development*, Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM, 2005, pp. 83–90.

[24] Tino Gramb, T Gram, and T Pellizzari, *Non-standard computation*, John Wiley & Sons, Inc., 1997.

[25] Ramin Halavati and Saeed Shouraki, *Symbiotic evolution to avoid linkage problem*, Linkage in Evolutionary Computation (2008), 285–314.

[26] M. Hartmann, P.K. Lehre, and P.C. Haddow, *Evolved digital circuits and genome complexity*, Evolvable Hardware, 2005. Proceedings. 2005 NASA/DoD Conference on, IEEE, 2005, pp. 79–86.

[27] Morten Hartmann and Pauline Catriona Haddow, *Evolution of fault-tolerant and noise-robust digital designs*, Computers and Digital Techniques, IEE Proceedings-, vol. 151, IET, 2004, pp. 287–294.

[28] R. Hassin and A. Levin, *A better-than-greedy approximation algorithm for the minimum set cover problem*, SIAM Journal on Computing **35** (2005), no. 1, 189–200.

[29] J.H. Holland, *Emergence: From chaos to order*, Oxford University Press, 2000.

[30] John H Holland, *Escaping brittleness: the possibilities of general purpose learning algorithms applied to parallel rule-based system*, Machine learning (1986), 593–623.

[31] G.S. Hornby, *Functional scalability through generative representations: the evolution of table designs*, Environment and Planning B **31** (2004), no. 4, 569–588.

[32] ______, *Properties of artifact representations for evolutionary design*, (2004).

[33] F. Jiménez Morales, J.P. Crutchfield, and M. Mitchell, *Evolving two-dimensional cellular automata to perform density classification: A report on work in progress*, Parallel Computing **27** (2001), no. 5, 571–585.

[34] S.A. Kauffman, *The origins of order: Self-organization and selection in evolution*, Oxford University Press, USA, 1993.

[35] Sanjeev Kumar and PJ Bentley, *The abcs of evolutionary design: Investigating the evolvability of embryogenies for morphogenesis*, Genetic and Evolutionary Computation Conf.(GECCO'99) Late Breakers, Orlando, Florida USA, 1999.

[36] George F Luger, *Artificial intelligence: Structures and strategies for complex problem solving*, Addison-Wesley Longman, 2002.

[37] Pradipta Maji, Chandrama Shaw, Niloy Ganguly, Biplab K Sikdar, and P Pal Chaudhuri, *Theory and application of cellular automata for pattern classification*, Fundamenta Informaticae **58** (2003), no. 3, 321–354.

[38] Jacques Mazoyer, *A six-state minimal time solution to the firing squad synchronization problem*, Theoretical Computer Science **50** (1987), no. 2, 183–238.

[39] ______, *An overview of the firing squad synchronization problem*, Automata Networks (1988), 82–94.

[40] Julian Miller and Peter Thomson, *A developmental method for growing graphs and circuits*, Evolvable Systems: From Biology to Hardware (2003), 93–104.

[41] Julian F Miller, Dominic Job, and Vesselin K Vassilev, *Principles in the evolutionary design of digital circuitspart i*, Genetic programming and evolvable machines **1** (2000), no. 1, 7–35.

[42] ______, *Principles in the evolutionary design of digital circuitspart ii*, Genetic programming and evolvable machines **1** (2000), no. 3, 259–288.

[43] Julian F Miller and Peter Thomson, *Cartesian genetic programming*, Lecture Notes in Computer Science (2000), 121–132.

[44] Julian F Miller, Peter Thomson, and Terence Fogarty, *Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study*, 1997.

[45] FR Moore and GG Langdon, *A generalized firing squad problem*, Information and Control **12** (1968), no. 3, 212–220.

[46] Can Öztürkeri and Colin G. Johnson, *Evolution of self-assembling patterns in cellular automata using development*, J. Cellular Automata **6** (2011), no. 4-5, 257–300.

[47] K. Preston and M.J.B. Duff, *Modern cellular automata: theory and applications*, vol. 198, Plenum Press London, 1984.

[48] Paul L Rosin, *Training cellular automata for image processing*, Image Processing, IEEE Transactions on **15** (2006), no. 7, 2076–2087.

[49] AF Rozenfeld, K. Laneri, and EV Albano, *Critical dynamic approach to stationary states in complex systems*, The European Physical Journal-Special Topics **143** (2007), no. 1, 3–8.

[50] LS Schulman and PE Seiden, *Statistical mechanics of a dynamical system based on conway's game of life*, Journal of Statistical Physics **19** (1978), no. 3, 293–314.

[51] Claude E Shannon, *Communication theory of secrecy systems*, Bell system technical journal **28** (1949), no. 4, 656–715.

[52] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer, *A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems*, Evolutionary Computation, IEEE Transactions on **1** (1997), no. 1, 83–97.

[53] N. J. A. Sloane, *The On-Line Encyclopedia of Integer Sequences*, A000110, Bell or exponential numbers: ways of placing n labeled balls into n indistinguishable boxes.

[54] A.R. Smith, *Real-time language recognition by one-dimensional cellular automata*, Journal of Computer and System Sciences **6** (1972), no. 3, 233–253.

[55] Stephen F Smith, *Flexible learning of problem solving heuristics through adaptive search*, Proceedings of the Eighth international joint conference on Artificial intelligence, vol. 1, Citeseer, 1983, pp. 422–425.

[56] WB Spillman Jr, T Zeng, and RO Claus, *Modeling the electro-static self-assembly process using stochastic cellular automata*, Smart materials and structures **11** (2002), no. 5, 623.

[57] M. Spivey, *A generalized recurrence for bell numbers*, Journal of Integer Sequences **11** (2008), no. 2.

[58] Kenneth O Stanley, *Compositional pattern producing networks: A novel abstraction of development*, Genetic Programming and Evolvable Machines **8** (2007), no. 2, 131–162.

[59] Kenneth O Stanley, Joseph Reisinger, and Risto Miikkulainen, *Exploiting morphological conventions for genetic reuse*, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004) Workshop Program. Springer Verlag, Berlin, 2004.

[60] Adrian Stoica, Didier Keymeulen, Raoul Tawel, Carlos Salazar-Lazaro, and Wei-te Li, *Evolutionary experiments with a fine-grained reconfigurable architecture for analog and digital cmos circuits*, Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on, IEEE, 1999, pp. 76–84.

[61] Adrian Stoica, Ricardo S Zebulum, Xin Guo, Didier Keymeulen, Michael I Ferguson, and Vu Duong, *Silicon validation of evolution-designed circuits*, Evolvable Hardware, 2003. Proceedings. NASA/DoD Conference on, IEEE, 2003, pp. 21–25.

[62] X. Sun, P.L. Rosin, and R.R. Martin, *Fast rule identification and neighborhood selection for cellular automata*, Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on **41** (2011), no. 3, 749–760.

[63] Eiichi Takahashi, Yuji Kasai, Masahiro Murakawa, and Tetsuya Higuchi, *A post-silicon clock timing adjustment using genetic algorithms*, VLSI Circuits, 2003. Digest of Technical Papers. 2003 Symposium on, IEEE, 2003, pp. 13–16.

[64] G. Tufte, *Phenotypic, developmental and computational resources: scaling in artificial development*, Proceedings of the 10th annual conference on Genetic and evolutionary computation, ACM, 2008, pp. 859–866.

[65] John Von Neumann and Arthur W Burks, *Theory of self-reproducing automata*, (1966).

[66] Eric W. Weisstein, *Cyclic group. from mathworld–a wolfram web resource*, December 2012.

[67] S. Wolfram, *Statistical mechanics of cellular automata*, Reviews of modern physics **55** (1983), no. 3, 601.

[68] ______, *A new kind of science*, Wolfram Media, Inc., Champaign, Illinois, 2002.

[69] Stephen Wolfram, *Cellular automata*, Los Alamos Science **9** (1983), 2–27.

[70] ______, *Cellular automata and complexity*, (2006).

[71] A. Wuensche, *Basins of attraction in network dynamics*, Modularity in development and evolution (2004), 1–17.

[72] Yingxu Yang and SA Billings, *Extracting boolean rules from ca patterns*, Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on **30** (2000), no. 4, 573–580.

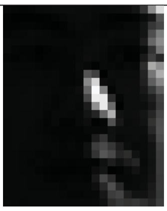
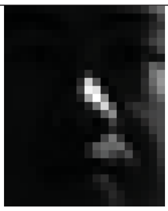
[73] Albert Y Zomaya, *Handbook of nature-inspired and innovative computing: integrating classical models with emerging technologies*, Springer, 2006.

Appendix A

Used Images From the Extended Yale Face Database B

A.1 The 20-by-25 images

Table A.1: The set of 20 × 25 images used in the experiments of Chapter 3

A.2 The 40-by-50 images

Table A.2: The set of 20 × 25 images used in the experiments of Chapter 3