To the Graduate Council:

I am submitting herewith a project report written by Robert A. Coop entitled "Functional Analysis of Cellular Automata." I have examined the final paper copy of this project report for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Itamar Arel, Major Professor

We have read this project report
and recommend its acceptance:

Dr. Gregory Peterson

Dr. Hairong Qi

Dr. James Nutaro

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

Functional Analysis of Cellular Automata

A Project Report Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Robert A. Coop

August 2010

Abstract

Cellular automata are systems that consist of a number of homogenous sub-systems (referred to as cells). Cellular automata models are used in many different disciplines and are capable of exhibiting many different types of physical, biological, or information-theoretic behaviors. Within a cellular automaton, each cell is associated with a particular state. A new generation of cells are created according to some fixed transition rule, where the next state of a cell is determined based on the state history of some set of cells in the system (referred to as that cell's neighborhood). For determinate systems, the state progression of a cellular automata is uniquely determined by its initial state, neighborhood function, and transition function. Given only the neighborhood function of a determinate cellular automaton, one can infer its transition function by observing the state transitions that take place and building a lookup table that defines the transition function. However, given only observations of the state transitions, it is difficult to infer the neighborhood function that governs the cellular automaton. This project implements a discrete event specification model of a general cellular automaton, uses this simulation to generate a number of different state transition trajectories, and uses an evolutionary algorithm to search the space of possible neighborhood functions in order to find the neighborhood function that was originally used to generate the trajectories. Results are presented that show the intelligent search algorithm is significantly more efficient than brute force search under most circumstances.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A cellular automaton is a system composed of a number of homogeneous sub-systems (referred to as cells). For a given cellular automaton, each cell is governed by the same neighborhood and transition functions; the neighborhood function determines the set of cells in a system whose state history has an effect on a given cell, and the transition function computes the next state of a cell, given the states of all cells in its neighborhood. It should be noted that the states of the cells in the neighborhood can be from the previous time step or can be from many time steps in the past. Cellular automata can model a large variety of different systems very efficiently. There are many different examples in the literature of modeling systems as cellular automata with beneficial effect [3, 6, 15].

Functional analysis of a system is the process of taking a set of observations of a system's behavior and trying to infer how components of that system affect each other [7]. It is a non-trivial problem, with no solution in general. Applied to cellular automata, the functional analysis task is equivalent to the task of identifying the neighborhood function that governs the automaton using only observations of the automaton in operation and no a-priori knowledge of the transition function or neighborhood function used to generate the system. There are many practical applications of an algorithm that can solve this problem. Cellular automata have

successfully been used to model diverse systems such as biological processes [5] and freeway traffic [11]. Given an efficient means of functionally analyzing cellular automata, it may be feasible to examine biological data in order to find cells within a tumor that affect the growth of the tumor as a whole, or to examine traffic patterns and identify intersections that have an affect on the overall level of traffic in the system.

There are two main goals for this project. The first goal is to create and implement a simulation of a very general cellular automaton as a discrete event system (DEVS). Discrete event systems consist of formal specifications of how a model behaves as it receives external input and time passes [16], and they can be very efficiently simulated by a number of different software packages (e.g. [12]). A general model of a cellular automaton is implemented as a DEVS system; this allows for the simulation of cellular automata regardless of details such as the dimensionality of the state that each of the cells has or the particulars of the transition or neighborhood functions that govern the automaton.

The second goal is to perform functional analysis on cellular automata by examining simulation trajectories of the automaton and inferring the neighborhood function that was used to create the system. An evolutionary algorithm based on population-based incremental learning [1] which is particularly effective at doing this is presented and implemented, and a number of tests to demonstrate the capabilities of the algorithm are presented.

This paper is divided into 9 different chapters. In chapter 2, we examine the discrete event system specification and provide an overview of the procedure used to simulate a system defined using DEVS. Chapter 3 provides a general description of the cellular automaton, including formalizing cellular automata as DEVS models. In chapter 4 we describe the implementation of the first project component, the simulation implementation. Chapter 5 presents the background of functional analysis and lays the groundwork for the search algorithm. We then cover the evolutionary search algorithm in chapter 6. The implementation for the search

algorithm is presented in chapter 7, and experimental results are presented in chapter 8. Conclusions and notes regarding future work are presented in chapter 9.

Chapter 2

Discrete Event System Specification

The Discrete Event System Specification (DEVS) formalism was used to model the behavior of the cellular automaton for this project. DEVS is a formalism that describes the behavior of models in a very general, robust fashion. Using this formalism, models can be represented and simulated very efficiently.

2.1 DEVS Formalism

Formally, a DEVS model M is defined [16] as a tuple:

$$M =< X, Y, S, ta, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda > \tag{2.1}$$

where

- X is the set of input events
- Y is the set of output events
- S is the set of possible states

- ta is the time* advance function $ta : S \to \mathbb{R}$ which, given the current state, provides the time when the next internal transition will occur
- δ_{int} is the internal transition function $\delta_{int} : S \to S$ which computes the next state that a model will transition to in the absence of external input
- δ_{ext} is the external transition function $\delta_{ext} : S \times X^b \times \mathbb{R} \to S$ (where X^b is a collection of input events) which maps the model's current state, external input events, and elapsed time since the last transition to the next state
- δ_{conf} is the confluent transition function $\delta_{conf} : S \times X^b \to S$ which is used to resolve the conflict that arises when the model receives external input at the same time that it would ordinarily undergo an internal transition
- λ is the output function $\lambda : S \to Y$ which provides the output of the current state when an internal (or confluent) transition occurs

Models within a simulation can be connected to each other, in which case the output from one model becomes the input for any connected models. Formally, in the case of connected models, each model's definition includes the set of couplings and translation functions that map the model's output set to the coupled model's input set. Because these details are not significant for this project, they have been omitted.

2.2 Simulating a model using DEVS

The process of simulating a DEVS model is fairly straightforward. Starting at time $t = 0$, the process is as follows:

*A note about time: Formally, DEVS uses a timebase $t \in (\mathbb{R}, \mathbb{Z})$. This is to allow for scheduling of state transitions and to make bookkeeping easier when manually calculating or analyzing simulation trajectories. For example, a model that received an input at time $t = (1, 0)$ that triggered a state transition would actually process this state transition at time $t = (1, 1)$ in order to simplify event scheduling. When using software to run simulations, this detail is usually hidden by the simulation software; for simplicity, we omit this detail in this project report and treat the timebase as if it is $t \in \mathbb{R}$.

1. The time of the next input (and the receiving model) is recorded as (M_{ext},t_{ext}). The time advance function (ta) of all models is computed, and the time of the next internal transition (and the associated model) is recorded as (M_{int}, t_{int}). The model with the lowest time is selected for processing. In the event that multiple models have identical times, they are processed sequentially.

2. For each selected model M, the time of the next internal transition (t_{int}) and the time of the next input event (t_{ext}) are compared, then there are two loops over the selected models that are performed.

3. The first loop over the models calculates the output for each M where $t_{int} \leq t_{ext}$ using the output function (λ).[†] Any output generated by M is immediately moved to any coupled models.

4. The second loop over the models computes the new state of each model.

   (a) If M has no external input, the internal transition function (δ_{int}) is used to compute the next state of M.

   (b) If M has external input and no internal transition, the external transition function (δ_{ext}) is used to compute the next state of M.

   (c) If M has external input and an internal transition, the confluent transition function (δ_{int}) is used to compute the next state of M.

5. The state of M is updated and the cycle repeats until there is no further external input to the simulation and all models have a time advance value of $ta = \infty$.

[†]Note that M is not given an opportunity to produce output in response to external inputs. In situations where the model must produce immediate output in response to external input, this is accomplished by transitioning to a transient state that immediately undergoes an internal transition and produces the desired output.

Chapter 3

The Cellular Automaton

The cellular automaton is a model of a system that is constructed of many homogeneous sub-systems. Cellular automata have a number of practical applications including representing thermodynamic systems, biological systems, and many other physical and informational phenomena. There are a number of papers and books that describe the different applications of cellular automata theory (e.g. [6, 15, 3]). For the purposes of this project, we are primarily concerned with being able to generally represent cellular automata and simulate the trajectory of a cellular automaton as it undergoes the various internal transitions determined by the rules that govern the system.

3.1 Defining Cellular Automata

A cellular automaton is a system that consists of a collection of homogeneous sub-systems (referred to as cells) that interact with each other. Each cell has an independent state, and at every time step each cell transitions to a new state based on its previous state and the state of (some) other cells in the system. Here we present a traditional formal definition of cellular automata, though it is worth noting that there are a large number of differing formal specifications of cellular automata (e.g. [14, 10, 4]).

We can view a cellular automaton as being a system composed of $K \in \mathbb{Z}$ individual cells. The state of a cell is represented by

$$a_i^{(t)} \in S \tag{3.1}$$

where $t \in \mathbb{Z}$ refers to the system time*, $i \in [0, K-1]$ refers to the cell being considered, and S is the set of all possible states that a cell could have. The overall state of the cellular automaton at time t is the cross-product of all of the cells within it ($\prod_{i=0}^{K-1} a_i^{(t)}$).

The cellular automaton's state changes at each time step according to some transition function F and neighborhood function N that is shared for each cell. So, the next state of cell i is given by

$$a_i^{(t+1)} = F(N(i,t)) \tag{3.2}$$

where F is the transition function of the form

$$F : S^M \to S \tag{3.3}$$

and N is the neighborhood function of the form

$$N : (i \in [0, K-1], t \in \mathbb{Z}) \to S^M \tag{3.4}$$

The neighborhood function specifies the set of cell states S^M that have an impact on the state of cell i at time $t+1$.

For example, in a system where the next state of cell i is only dependent on the previous state of cell i, the neighborhood function could be defined as

$$N(i,t) = a_i^{(t)} \tag{3.5}$$

*In this project we consider only discrete-timed systems.

in which case $M = 1$, and $F : S \rightarrow S$.

In a system where the next state of a system is dependent on the previous state of that cell and all cells within a radius r, the neighborhood function could be defined as

$$N(i,t) = \prod_{k=(i-r)}^{i+r} a^{(t)}_{(k \bmod K)} \tag{3.6}$$

(where $\prod$ refers to the cross-product) in which case $M = 2r + 1$.

If such a system had longer term temporal dependencies, and each cell's next state depended on the s previous states of each cell within a radius r, the neighborhood function could be defined as

$$N(i,t) = \prod_{j=t-s}^{t} \left[\prod_{k=(i-r)}^{i+r} a^{(j)}_{(k \bmod K)} \right] \tag{3.7}$$

in which case $M = s(2r + 1)$.

It should be noted that, in these examples, the neighborhood of each cell consists of some contiguous block of cells determined by a temporal depth s and a spatial radius r. Under systems where this is the case, determining the neighborhood is only a matter of identifying s and r. In general, this cannot be assumed to be the case. In the general case, one must identify each individual member of the neighborhood; this adds a great deal of complexity to the neighborhood search problem.

3.2 DEVS Formalism

In expressing cellular automata in terms of the DEVS formalism, it is most convenient to treat each cell as an individual model. Because of the uniformity of the simulation, the actual model is fairly straightforward (as far as DEVS models go). For this project, two differing formalisms were developed. The first version conforms more to the DEVS conventions with regard to input and output message passing; it proved too

Table 3.1: Original DEVS Cell Model

Each cell is defined as a DEVS model $C_i =< X, Y, S, ta, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda >$ where

- $X = \{s|s \in S\}^M$, where M is the number of cells in the neighborhood of C_i
- $Y = \{s|s \in S\}$
- $S \subset \mathbb{Z}^D$, where D is the dimensionality of the state vector, and each dimension can take on one of a fixed number of possible values.
- $ta = 1$
- $\delta_{int} = \delta_{ext} = \{\}$. Due to the fact that each cell always generates an output and receives an external message from each cell in its neighborhood, only δ_{conf} is ever triggered.
- δ_{conf} : When δ_{conf} is triggered, the cell has an input X^b that contains the state of all cells in the neighborhood of C_i (i.e. $X^b = N(i,t)$). The cell then computes its transition function and updates its current state such that $a_i^{(t+1)} = F(X^b)$.
- $\lambda = a_i^t$, as each cell broadcasts its current state at each time step. Note that the output of a cell is only delivered to cells C_k that contain the broadcasting cell in their neighborhood such that C_i is in the neighborhood $N(k,t)$.

complex when dealing with systems that have variable-length temporal dependencies. Both of these DEVS models are described here.

3.2.1 Original Formalism

The original approach to modeling cellular automata was to define each cell C_k as a model that is coupled to all cells that include C_k in their neighborhood (i.e. all cells C_i such that C_k is in the neighborhood $N(i,t)$). The time advance for a cell is a fixed interval, and, due to the fact that each cell generates an output at each time step, the only transition function that actually gets used is the confluent transition function. The formal definition of this model can be found in table 3.1.

Table 3.2: Simplified DEVS Cell Model

Each cell is defined as a DEVS model $C_i =< X, Y, S, ta, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda >$ where

- $X = Y = \{\}$
- $S \subset \mathbb{Z}^D$, where D is the dimensionality of the state vector, and each dimension can take on one of a fixed number of possible values.
- $ta = 1$
- $\delta_{ext} = \delta_{conf} = \{\}$
- $\lambda = \{\}$
- δ_{int} : When δ_{int} is triggered, the cell looks up the state of all cells within its neighborhood in the system table. The cell then computes its transition function and updates its current state such that $a_i^{(t+1)} = F(N(i,t))$, and records the new state in the system table.

3.2.2 Relaxed Formalism

The original formalism required that each cell keep track of the state of all cells in its neighborhood. The states of the cells in the neighborhood were received as external messages at each time step. Each cell also had to record which cells contained it as a member in their neighborhood in order to broadcast its state. Due to this, the bookkeeping for each cell quickly became a problem when considering neighborhoods with temporal dependencies that spanned multiple time steps.

To address this problem, the DEVS model was relaxed slightly in that an external table of the states of all cells in the system was created. The table recorded the state of all cells in the system for as many time steps as needed; for example, if the neighborhood for a cell contained the state of a cell n time steps ago then the table would record the current state of the system as well as the previous n states. This greatly simplified the model of a cell, as there was no need to actually have the model output its state as long as the state was recorded in the system table. In addition, due to the fact that cells were no longer passing messages back and forth, only internal transitions would occur. The simplified DEVS definition can be found in table 3.2.

Chapter 4

Simulation Implementation

In this project, the goal was to design and implement a system for simulating cellular automata in general. Though the author did create a customized DEVS simulation engine, it ended up being more prudent to use a pre-existing software library which the author has worked with in the past. The simulation framework was created using the ADEVS C++ library [12]. In implementing the cellular automata simulation, a general representation of a cell, a cell space that contains multiple cells, a cell's state, neighborhood, and transition function was created. Implementation and technical details are given here for these components and the simulation process.

4.1 Generalized Cell Model

The generalized cell model was created by sub-classing the ADEVS Atomic class, which represents an atomic DEVS model. The GeneralCell class is responsible for containing the state of a cell (further state detail is provided below in section 4.3) and contains the implementation of the DEVS-related functionality for a cell (specified in table 3.2). The DEVS functions are called during a simulation by the ADEVS simulation engine.

4.2 Generalized Cell Space

The generalized cell space was created by sub-classing the ADEVS CellSpace class. The GeneralSpace class is responsible for containing each GeneralCell used in the cellular automaton, tracking the state of the automaton (with the overall automaton state being the cross-product of the state of each individual cell), and recording the history table used for updating cell states (see section 3.2.2). The GeneralSpace implementation is fairly simple and primarily deals with maintaining the history table. The ADEVS CellSpace class handles the functionality required for a GeneralSpace to contain multiple GeneralCell models.

4.3 Generalized State

To ease the process of simulating cellular automata with different possible state representations, a generalized state class GeneralState was created and used in this project. The GeneralState class is able to represent any state space made up of a discrete number of dimensions that contain a discrete set of arbitrary values. Using this class, one is able to define the dimensionality of a state to be any $d > 0 \in \mathbb{Z}$, the size of each dimension to be any $n_d > 0 \in \mathbb{Z}$, and the possible values of that dimension to be any set of values $s^d = \{s_0^d, s_1^d, ..., s_{n_d-1}^d\}, s_i^d \in \mathbb{Z}$.

This is internally handled by keeping track of d, creating an array of d integers to track n_d, and creating an array of d arrays of integers to track the actual values possible for each state. These values are static and shared by each instance of a GeneralState. Each cell then contains a GeneralState which has an array of d integers $S_i, 0 < i \in \mathbb{Z} < d - 1$, where $S_i = k$ indicates that the i-th dimension of the cell's state has a value of s_k^i.

4.4 Generalized Neighborhood

Equation 3.4 describes the neighborhood function of a cellular automaton. This neighborhood specifies, for a given cell i at a given time t (denoted by $a_i^{(t)}$), what cell states act as inputs to the transition function that determines the value of $a_i^{(t+1)}$. If a cell k's state at time s is in the neighborhood of $a_i^{(t)}$, then the state is contained in the neighborhood function such that $a_k^{(s)}$ is in the neighborhood $N(i,t)$. It should be noted that a neighborhood function is specified for the entire cellular automaton and shared by each cell; the neighborhood does not differ from cell to cell within the same automaton. To allow for representing cellular automata in general, we must be able to represent any possible neighborhood.

The GeneralNeighborhood class accomplishes this. For a general neighborhood to be represented, one must specify the maximum possible spatial radius Δd and maximum possible temporal distance Δt such that $a_{i\pm\Delta d}^{(t-\Delta t)}$ is in the neighborhood $N(i,t)$. Once that is specified, we can represent the neighborhood function N as a matrix with Δt rows and $(2\Delta d+1)$ columns. This matrix contains a 1 for all entries N_{jk} where $N_{jk} = 1 \rightarrow a_{i-\Delta d-1+j}^{(t-j)}$ is in the neighborhood $N(i,t)$. This is similar to how functional masks are treated (discussed later in section 5.1).

For example, using a temporal depth of $\Delta t = 4$ and a spatial radius of $\Delta d = 4$, the matrix

$$N = \begin{array}{c|ccccccccc} & i-4 & i-3 & i-2 & i-1 & i & i+1 & i+2 & i+3 & i+4 \\ \hline t & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t-1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ t-2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ t-3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{4.1}$$

would indicate that $a_i^{(t+1)} = F(a_{i-4}^{(t)}, a_i^{(t-1)}, a_{i-2}^{(t-2)}, a_{i-4}^{(t-3)})$.

This is implemented in GeneralNeighborhood by specifying Δd and Δt, then using an array of boolean values to track which cells are in the neighborhood. The two

dimensional matrix N is flattened into a one-dimensional array with $(2\Delta d + 1)\Delta t$ entries.

4.5 Generalized Transition Function

To calculate the trajectory of a cellular automaton, we must have a transition function $F : S^M \rightarrow S$ that computes the next state of a cell $a_i^{(t+1)}$ from the M states in the neighborhood $N(i, t)$. This is accomplished through the GeneralTransitionFunction class. The purpose of this class is to generate a random valid transition function F that computes a resulting GeneralState given M GeneralStates as input.

This is implemented by creating a large lookup table recursively. To create a general transition function that maps M GeneralStates consisting of D dimensions of size n_d, the following procedure is used:

1. The class is initialized with an array of void pointers $function\_table = NULL$, the current input $i = 0$, and the current dimension dimension $d = 0$.

2. The table is calculated using the $create\_func(input, dimension)$ function such that $function\_table = create\_func(i, d)$.

3. $create\_func(i, d)$ performs the following computations:

   (a) If this is not the final recursive iteration (i.e. $i < M - 1$ or $d < D - 1$):

      i. An array of void pointers $vptr$ is allocated with room for n_d pointers.

      ii. For each possible input dimension value $k \in [0, n_d - 1]$, $vptr[k]$ is set recursively:

         A. If this is the final dimension for the current input (i.e. $d = D-1$), $vptr[k] = create\_func(i + 1, 0)$

         B. Otherwise, $vptr[k] = create\_func(i, d + 1)$

   (b) If this is the final recursive iteration (i.e. $i = M - 1$ and $d = D - 1$):

i. An array of void pointers $vptr$ is allocated with room for n_d pointers.

ii. Each pointer $vptr[k], k \in [0, n_d - 1]$ is set equal to a new randomly created GeneralState

(c) $vptr$ is returned as the output of $create\_func$

In this fashion, a lookup table is created. To calculate F, the process is initialized by setting $vptr = function\_table$. Then, the pointer corresponding to the value of each dimension (for each input in the M inputs) is dereferenced such that $vptr = vptr[k]$, where k is the index of the value for the current input and dimension. Once all of the inputs and dimensions have processed, $vptr$ will point to a GeneralState s such that $F(S_0, S_1, ..., S_{M-1}) = s$.

4.6 Simulation Process

To generate a cellular automaton and simulate system trajectories, the following process is used:

1. The program is initialized with the spatial radius Δd, temporal depth Δt, number of cells (M) in the neighborhood (such that $N = S_0 \times S_1 \times ... \times S_{M-1}$), and a seed for the random number generator being provided by the user. (In the project's implementation, such data is provided using command line arguments.)

2. The GeneralState dimensionality D, the size of the each dimension n_d, and the possible values of each dimension $s^d = \{s_0^d, s_1^d, ..., s_{n_d-1}^d\}$ are initialized using user-specified parameters. (In the project's implementation, such data is provided in a header file and compiled into the program.)

3. A random GeneralNeighborhood containing M cells is generated.

4. A GeneralSpace is created that contains all cells in the cellular automata (the number of which is specified via a header file), and GeneralCells are instantiated and added to the GeneralSpace.

5. An ADEVS simulation engine is created, and the GeneralSpace added to it.

6. The settings for the simulation and the generated GeneralNeighborhood are saved for future analysis*.

7. A user-specified number of system trajectories are calculated and saved according to the following procedure:

   (a) The state of each cell is randomly initialized.

   (b) Δt time steps are simulated without any state transitions in order to build up the required history for the cell neighborhood and transition function.

   (c) A user-specified number of time steps are simulated by calculating the next state of each cell using the cell neighborhood and transition function.

   (d) After each time step, the state of the system is saved for future analysis.

This procedure was used to generate the data used in the next section of the project, which seeks to infer the functional relationships between cells (equivalent to inferring the neighborhood definition for the cellular automata).

*The transition function specification was originally saved also, but for large neighborhoods and state definitions this proved to use too much disk space, so it is omitted. Note that it can be regenerated by initializing the simulation with the same parameters and random seed.

Chapter 5

The General Systems Problem Solver

The goal of the second component of this project was to take the cellular automaton trajectories that were simulated and infer the functional relationships that exist within the cellular automaton. Specifically, the goal is to be able to determine what neighborhood function N is being used by the automaton using only observations of the system trajectories. Due to the fact that the cellular automata created are deterministic, once the neighborhood function is known, the transition function F that determines the next state of each cell $a_i^{(t+1)} = F(N(i,t))$ can be calculated by observing the value of all the states in $N(i,t)$ and the resulting state $a_i^{(t+1)}$ and building a lookup table.

This component of the project was largely inspired by Klir's treatment of general systems theory (covered in [7]) and the research in fuzzy inductive reasoning that was later inspired by it (covered in [2]).

5.1 Functional Masks

The concept of a functional mask is identical to the neighborhood of a cell in a cellular automaton, but applied to a more general system. A functional mask for a system specifies the relationship between system variables by specifying which variables are inputs and which are outputs that functionally depend on the values of

those inputs. Using the same neighborhood from the example presented in equation 4.1, the equivalent functional mask could be represented as

$$M = \begin{array}{c|ccccccccc} & i-4 & i-3 & i-2 & i-1 & i & i+1 & i+2 & i+3 & i+4 \\ \hline t+1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ t & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t-1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ t-2 & 0 & 0 & -3 & 0 & 0 & 0 & 0 & 0 & 0 \\ t-3 & -4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{5.1}$$

where the numbers $n > 0$ indicate the n-th functional output, and the numbers $m < 0$ indicate the m-th functional input. This expresses the same relationship as the neighborhood from before; functional masks are just more general in that they can represent multiple outputs and apply to systems in general instead of just cellular automata. For the purposes of this project, we can disregard this distinction and represent cellular automata neighborhoods and functional masks in the same fashion. Notationally, we use $M(i,t)$ to refer to the set of cells that are treated as functional inputs by mask M for the state of cell i at time $(t+1)$ (in the same manner that $N(i,t)$ refers to the true set of cells that functionally determine the next state of cell i).

5.2 Possible Masks for a System

The problem that this project seeks to solve is that of inferring the functional mask that governs a cellular automaton given only simulated trajectories of the automaton. This is a non-trivial problem; when presented with only trajectories of the system, we have no knowledge of the spatial radius and temporal depth of the neighborhood used, nor do we know how many members are contained within the neighborhood.

For this project, we assume that we can place an upper bound on the spatial radius and temporal depth of the masks we are willing to consider. If we constrain our mask

search to masks of spatial radius d and temporal depth t, there are $M = (2d+1)t$ potential mask members for the system. In this case, the number of possible masks in this space is given by

$$C = \sum_{m=1}^{M} \frac{(M!)}{(m!)(M-m)!} \tag{5.2}$$

For example, a system using $d = 4$ and $t = 4$ (which we investigate further in sections 8.3.1 and 8.3.2) has $M = 36$ potential members and $C = 68,719,476,735$ possible masks to choose from.

5.3 Evaluating Functional Masks

Given a functional mask M, we must be able to determine if the mask correctly identifies the functional inputs and outputs for a given system. Specifically, when using cellular automata, we want to know if M correctly captures the neighborhood information that the cells use. If M is the correct mask for a cellular automaton, then the next state of cell i is determined by the relationship $a_i^{(t+1)} = F(M(i,t)) = F(N(i,t))$ which indicates that $M(i,t) = N(i,t)$.

M is tested by iterating over the recorded simulation trajectories for the system and determining if this relationship is functionally determinate. This is accomplished by using the following test procedure:

1. Select a simulation trajectory, a simulation time $t \geq \Delta t$, and a cell i.

2. Record set of functional inputs given by $I = M(i,t)$, and the resulting state of cell i given by $O = a_i^{(t+1)}$. The functional relationship we are checking is given by $F(I) = O$. We keep a table of the different values of I that have been seen, and note the value of O that was associated with I.

3. Depending on if I has been seen before, the test has the following results

   (a) If I has not been seen before, we record $I \rightarrow O$ in the table and consider the test a success.

(b) If I has been seen before, and it led to the same resulting state O, the test is considered a success.

(c) If I has been seen before, but it led to a different resulting state $O_1 \neq O$, the test is considered a failure. $I \rightarrow O$ is still recorded in the table, if it is not already present.

To test a mask, we perform the above test on some or all of the available history and keep track of how many successes and failures occur, as well as the frequency that an input pattern I led to an output pattern O.

In this project, many different heuristics were used for comparing how one mask performed compared to another. The heuristic that was eventually selected combines the amount of the Shannon entropy of the system is that is eliminated by the mask (the entropy reduction) and the frequency with which observations occur (the observaion ratio) [2].

The Shannon entropy of one input I is

$$H_I = \sum_{\forall O} p(O|I) \log_2 p(O|I) \tag{5.3}$$

with the summation operating over each O that was associated with a particular I, and $p(O|I)$ is the conditional probability of output O occurring, given that input I is observed. This probability is statistically estimated as

$$p(O|I) = \frac{\#\text{ of observations of I, then O}}{\#\text{ of observations of I}} \tag{5.4}$$

The overall entropy of the mask is then

$$H_M = -\sum_{\forall I} p(I) H_I \tag{5.5}$$

where $p(I)$ is the probability of an input occurring, statistically estimated as

$$p(I) = \frac{\#\text{ of observations of I}}{\#\text{ of observations}} \tag{5.6}$$

The maximum possible entropy occurs when all outputs have the same probability and is given by

$$H_{\max} = -\sum_{\forall I} p(i) \sum_{\forall O} \frac{1}{n_I} \log_2 \frac{1}{n_I} = -\sum_{\forall I} p(i) \log_2 \frac{1}{n_I} \tag{5.7}$$

where n_I refers to the number of outputs O that are associated with a particular I. The entropy reduction of a mask is then calculated as

$$H_r(M) = \begin{cases} 1.0 - \frac{H_M}{H_{\max}} & H_{\max} > 0 \\ 1 & H_{\max} = 0 \end{cases} \tag{5.8}$$

which gives the proportion of the entropy in the system observations eliminated by the mask if the system has any entropy (i.e. $H_{\max} > 0$), and is identically equal to 1.0 in the case that the system observations are determinate. Note that for the purposes of evaluating a mask M, the 'observation' of a system is equivalent to the state of all input cells for the mask (i.e. the observation $= M(i,t)$). The correct mask $M(i,t) = N(i,t)$ will always have an entropy reduction of 1.0 due to the fact that the observations under this mask are necessarily determinate, but the entropy reduction is still a useful quantity when trying to select the 'better' of two masks that are not the correct mask. $H_r(M)$ is a measure of how well M represents a determinate functional relationship from the input to the output.

The other term involved in calculating the quality of a mask measures how often observations occur. In cluster analysis, it has been asserted that each legal discrete state should be observed at least five times in order to be considered statistically valid [8]. Based on this, an observation ratio O_r is used as an additional contributor to the overall quality measure [9]:

$$O_r(M) = \frac{5n_{5+} + 4n_4 + 3n_3 + 2n_2 + n_1}{5n_{leg}} \tag{5.9}$$

where

- n_{leg} is the total number of legal observations
- n_1 is the number of observations that occurred only once
- n_2 is the number of observations that occurred twice
- n_3 is the number of observations that occurred three times
- n_4 is the number of observations that occurred four times
- n_{5+} is the number of observations that occurred five or more times

So, if every observation occurs at least 5 times, $O_r(M) = 1$. If each legal observation only occurs once, $O_r(M) = \frac{1}{5}$. O_r can be used as a measure of how well represented the input of M is in the observed data.

Using these two measures, we can present a measure of the overall quality of a mask:

$$Q(M) = H_r(M) * O_r(M) \tag{5.10}$$

This measure can be used to compare different candidate masks, where the optimal mask for a system is the one with the largest $Q(M)$ value.

Chapter 6

Intelligent Mask Search using Evolutionary Algorithms

6.1 Overview of Evolutionary Algorithms

Evolutionary algorithms (often referred to as genetic algorithms) are a class of algorithms which optimize a function by iteratively testing a number of candidate solutions and using the performance of those solutions to direct the creation of the next group of candidate solutions for testing. The general form of an evolutionary algorithm is presented in table 6.1.

6.2 Population Based Incremental Learning

Population-based incremental learning (PBIL) is an evolutionary algorithm that represents the population of possible solutions as a single vector of probabilities. This differs from traditional evolutionary algorithms that maintain multiple unique population members and has a number of advantages and disadvantages over traditional approaches. Traditional evolutionary algorithms perform crossover and mutation operations directly on individual population members; as a result, the

Table 6.1: Evolutionary Algorithm Procedure

In general, an evolutionary algorithm seeks to optimize some function F by finding an approximate solution $\hat{x}$ such that $\hat{x} \approx argmin_x F(x)$ (or $\hat{x} \approx argmin_x - F(x)$ if the intent is to maximize F), where $argmin_x F(x) \triangleq \{x | \forall_y F(y) \geq F(x)\}$.

1. An initial population P_t of candidate solutions $x_{1,2,..,N-1}$ is generated for generation $t = 0$.

2. Each population member is associated with a fitness $f_{x_i} = F(x_i)$. The member with the best fitness (with 'best' indicating the highest or lowest fitness, depending on whether the goal is maximization or minimization of F) seen so far is recorded as $\hat{x}$.

3. A number of population members are selected to contribute to the next generation according to the relative fitness of each member.

4. The selected members are used to generate the next population P_{t+1} using the recombination and mutation operations.

5. Recombination occurs by selecting some number of population members (the 'parents') and generating a new population member (the 'child') according to the traits and fitness of the selected parents.

6. Mutation occurs by randomly changing the traits of a child. Typically mutation only occurs with some low probability.

7. If $\hat{x}$ is an acceptable solution (determined by some pre-defined tolerance level) then the algorithm terminates. Otherwise, execution continues by looping back to step 2.

population has some difficulty maintaining a stable representation of valid solutions. For further comparison of traditional evolutionary algorithms (and a much more detailed description of the PBIL algorithm) readers can refer to [1].

6.2.1 Overview of Algorithm

PBIL represents a population with a single vector of probabilities P. For a mask with M possible members, the vector will have a probability for each possible member (i.e. $P \in [0,1]^M$); these entries specify the probability with which a population member will have a particular bit set. A complete description of how the PBIL algorithm proceeds is presented in table 6.2.

6.2.2 Approach

PBIL was applied to the mask search problem in the following fashion. The goal of the search problem is to find the best possible functional mask that represents the neighborhood used to define a cellular automaton, given only a number of simulation trajectories created by that cellular automaton. To do this, a maximum allowable spatial radius r and maximum allowable temporal depth d was specified; a population member X_i for PBIL was defined as a $(2r+1)*d$ length binary array, where X_i is a flattened representation of the neighborhood matrix with $X_i(j) = 1$ indicating that a cell's state is part of the neighborhood.

The PBIL algorithm was then run by generating a number of population members X_i from the probability vector P, evaluating the fitness of each population members (according to the process described in section 5.3), adjusting P, and repeating the process. Theoretically, we would expect to see that the population members that contained entries $X_i(j)$ that correspond to the actual neighborhood N would have a higher fitness, and therefore the values of P that correspond to the entries in N would take on a higher probability than the other entries. The end result is that it would become more and more likely for a population member to be generated such that it

Table 6.2: PBIL Algorithm

1. Each element of P is initialized to some initial probability p_{init}.

2. A generation of population members is created according to equation

$$X_i = \begin{cases} 1 & n_i < P_i \\ 0 & otherwise \end{cases} \tag{6.1}$$

   where $i \in [0, M-1]$ and n is a $M \times 1$ vector of uniformly distributed random numbers in the range $[0, 1]$.

3. The fitness of each population member is calculated according to equation 5.10

4. The population member with the highest fitness X^b and the member of the population with the lowest fitness X^w are recorded. If X^b is the most-fit population member seen thus far, it is recorded as the current best solution $\hat{X}$. (Note that if all members of the population have an identical fitness value, the modification described in section 6.2.3 and table 6.3 comes into play.)

5. Recombination is performed by adjusting P such that each bit that differs between X^b and X^w is made more or less likely according to the following formula

$$P_i = \begin{cases} (1-\alpha)P_i + \alpha & X_i^b \overline{X_i^w} \\ (1-\beta)P_i & \overline{X_i^b} X_i^w \\ P_i & otherwise \end{cases} \tag{6.2}$$

   where $\alpha \in [0, 1]$ is the positive learning rate, $\beta \in [0, 1]$ is the negative learning rate, and $X_i^a \overline{X_i^b}$ indicates that bit i is present in X^a and absent in X^b.

6. Mutation is performed by randomly altering the values of some elements of P. For each element P_i, mutation is performed by applying the following equation

$$P_i = \begin{cases} P_i & n_i > \delta \\ (1-\gamma)P_i & \frac{1}{2}\delta < n_i \leq \delta \\ (1-\gamma)P_i + \gamma & 0 \leq n_i \leq \frac{1}{2}\delta \end{cases} \tag{6.3}$$

   where $\delta \in [0, 1]$ is the rate that mutations occur, $\gamma \in [0, 1]$ is the degree that a mutated value changes, and $n_i \sim U(0, 1)$ is a random value uniformly distributed in the $[0, 1]$ range. This equation has the effect of mutating a value with the probability δ and, of the mutations that occur, half will cause an increase in probability proportional to γ and half will cause a decrease in probability proportional to γ.

7. If $\hat{X}$ is an acceptable solution (determined by some pre-defined tolerance) then the algorithm terminates. Otherwise, execution continues by looping back to step 2.

exactly matches N. In chapter 8 a number of experiments based on this procedure are performed and analyzed.

6.2.3 Improving the PBIL Algorithm

A slight modification of the PBIL algorithm was developed for this project. In the algorithm described in table 6.2, the probability vector P is adjusted each generation according to the bits that differ between the best and the worst population member from that generation. It was discovered that, for this particular problem, there are many instances where an entire generation of population members can have an identical fitness value. Typically, this would occur if all of the potential functional masks being tested were too large; the size of the mask causes it to be very unlikely that there will be functional conflicts and can cause all of the population members to have identical fitness values. The solution to this particular problem was to introduce a penalty adjustment that occurs in the event that the fitness values for a generation are identical. In this case, P is adjusted such that more variation between the population members is introduced. Further detail about the penalty adjustment is given in table 6.3.

6.2.4 Drawbacks

One primary disadvantage of using the PBIL algorithm for functional mask searching was identified. The PBIL algorithm generates candidate solutions to the mask search problem (for masks of max possible length n) by generating a vector of n random numbers uniformly distributed in the range $[0, 1]$. Depending on whether each random number is less than P_i or not, the candidate solution will have bit i set or unset. For small to medium size masks, this method is quite effective. However, for larger masks, a problem with the PBIL algorithm becomes apparent. In particular, this algorithm does not perform well in the case that there is a large number of possible mask members n and a comparatively small number of actual mask members m.

Table 6.3: PBIL Penalty Adjustment Modification

During the PBIL algorithm described in table 6.2, a special adjustment procedure occurs whenever there is a generation of solution candidates that have an identical fitness value. The goal is to adjust P such that more variation is introduced into the population. This is accomplished by adjusting each element of P_i according to the formula

$$P_i = \begin{cases} (1-\phi)P_i & \forall_j X_i^j \\ (1-\phi)P_i + \phi & \forall_j \overline{X_i^j} \\ P_i & otherwise \end{cases} \tag{6.4}$$

where $\forall_j X_i^j$ indicates that all population members have bit i activated, $\forall_j \overline{X_i^j}$ indicates that all population members have bit i deactivated, and $\phi \in [0, 1]$ is a penalty adjustment factor. This has the effect of making bits that are active in all population members less likely to occur in the next generation, making bits that are inactive in all population members more likely to occur in the next generation, and leaving all other bits unchanged.

We can see why this occurs by examining the case where we are trying to find a mask with a maximum spatial radius of $r = 9$, a maximum temporal depth of $d = 9$, and $m = 19$ members (experimental results from this case can be found in section 8.3.4). In this case, the number of possible mask members $n = (2r + 1)d = 171$. The population probability vector therefore has 171 entries. Since the population of candidate masks is generated by randomly setting and unsetting bits according to the probability vector, we can calculate the expected number of bits that will be set for a population member by the formula

$$E[length] = \sum_{i=0}^{n} \left[i \begin{pmatrix} n \\ i \end{pmatrix} p^i (1-p)^{n-i} \right] \tag{6.5}$$

where $\begin{pmatrix} n \\ i \end{pmatrix}$ is the binomial coefficient, and each probability in P is equal to p. If $p = 0.5$, $E[length] = 85.5$, if $p = 0.25$, $E[length] = 42.75$, if $p = 0.1$, $E[length] = 17$, and if $p = 0.01$, $E[length] = 1.71$.

This gives us a bit of intuition regarding how the PBIL search algorithm behaves for large numbers of possible mask members; assuming we start with $P = P_{init} = 0.5$ in this case, we can expect that the average candidate solution will contain 86 neighborhood members. Considering that we use the process described in section 5.3 to evaluate these candidate solutions, we can expect that the algorithm will not perform very well if the actual neighborhood for a system is very small compared to the number of potential members of the neighborhood. If we test a potential mask M by computing $a_i^{(t+1)} = F(M)$ for all the available simulation history that we have and checking for functional conflicts, we can see that defining $F : S^{86} \rightarrow S$ is unlikely to have any conflicts even though the mask is not very optimal. So, from the very beginning of the algorithm, we can expect that masks are likely to have similar fitness values regardless of whether they contain the correct entries or not.

Even if we disregard the fact that the selectivity (the tendency of the algorithm to prefer correct masks over incorrect ones) of the algorithm is poor, we can see another problem arise with large mask sizes. Recall that the algorithm generates random candidate solutions probabilistically according to the vector P. The only way for the algorithm to find the single correct solution is for it to be randomly generated from a realization of P. If all non-correct mask entries have a probability of p_0 and all correct mask entries have a probability of p_1, the probability that the correct solution will be generated is expressed by

$$p(correct\ mask) = p_1^m (1 - p_0)^{n-m} \tag{6.6}$$

From successful tests (e.g. those in sections 8.3.1, 8.3.2) we know that a typical end-of-test value for the average correct mask entry is $p_1 = 0.5$ and a typical value for other mask entries is $p_0 = 0.15$. However, for a test with $n = 171$ and $m = 19$, this gives a $p(correct\ mask) = 3.57 \times 10^{-17}$; calculating the expected number of masks

N_M that must be generated before the correct one is generated as

$$E[N_M] = \sum_{i=1}^{\infty} \big(i * (1 - p(correct\ mask))^{i-1} * p(correct\ mask)\big) \tag{6.7}$$

we see that for these values $E[N_M] \approx \infty$! If we let $p_1 = .9$ and $p_0 = .1$, then we obtain $E[N_M] = 66,700,000$. This indicates that, on average, we'd need to generate approximately 67 million masks at those probability levels before the correct mask would be generated. However, this is impossible to do in practice, as the probability levels change after each generation. For the parameters used in this project, that means the probability levels will change after every 20 masks generated.

So, we can see that one significant weakness of this algorithm is the case where we have a large number of potential mask members but a small number of actual mask numbers.

Chapter 7

Mask Search Implementation

In this section, the details of the implementation of the procedures discussed in chapters 5 and 6 are discussed.

7.1 Processing Simulation Data

Chapter 4 detailed the process of implementing and simulating systems of cellular automata. This section discusses the process of reading and interpreting the data recorded during those simulations. For practical reasons, the the implementation of this portion of the project was treated as being completely independent of the previous portion; the code was independently created in order to allow for using the mask search implementation to process data created from other sources.

The fundamental class that handles the simulation data is the SimulationInstance class. Given the details of the system size, neighborhood radius, neighborhood depth, number of members in the neighborhood, and the random seed used to generate the data, this class handles the task of finding and loading the data for each of the simulation trajectories that were created for that system. (The details are needed primarily to identify the directory where the data is stored and the structure of the data files.) Having located all the sample files, each separate simulation trajectory that was generated (stored in separate flat files) is associated with a

SimulationTrajectory class, which in turn manages loading and reading the data for that simulation trajectory. Each simulation trajectory is treated as a series of SystemObservation elements; a SystemObservation consists of the state of each cell in the system at a given time.

7.2 The System Mask

The SystemMask class was developed to handle the concept of a functional mask. A SystemMask is specified by a spatial radius and a temporal depth; keeping track of which cell states are and are not part of the functional mask is handled in the same fashion as general neighborhoods (covered in section 4.4). A system mask is always assumed to be applied to a given element at a given time, and specifies the membership relative to that element and time. Given a SimulationTrajectory, it is possible to apply a SystemMask to any element at any time equal to or greater than the mask's temporal depth. Applying a SystemMask in this fashion will yield a special SystemObservation, where the 1st element of the SystemObservation is the state of the element at that time, the 2nd element is the state of the 1st functional input specified by that mask, the 3rd element is the state of the 2nd functional input, the 4th element is the state of the 3rd functional input, etc.

7.2.1 Determining the Quality of a System Mask (The FunctionHashTable)

To compute the quality of a mask (defined by equation 5.10), a mask is tested over some number of different input/output mappings in order to determine the entropy reduction of the mask, observation ratio of the mask, and if there are any functional conflicts caused by the mask. A functional conflict is defined as a situation where one set of input states is observed to lead to more than output state. To facilitate this testing process, and in anticipation of the fact that the testing process is likely

to be the bottleneck of the mask search procedure, it is desirable to have an efficient method of recording the different input/output mappings that occur and detecting functional conflicts. To accomplish this, the FunctionHashTable class was created.

Each SystemMask class has its own FunctionHashTable. The FunctionHashTable implements a slight variation of Pearson's hash algorithm [13]. For each SystemObservation that represents an input/output mapping, a hash value is calculated from the values of all the functional inputs in the mask. It is important to note (and was discovered to be a bug present in early project implementations) that one must only calculate the hash value of the functional inputs. If the functional output is also used in the hash calculation then the ability to detect functional conflicts is severely hampered by the fact that conflicting entries will most likely be stored in separate locations in the hash table, and the conflict will be undetected. Once the hash value is computed from the functional inputs, the value is used as an index into an array of singly-linked lists. If there is not already an entry for that particular input pattern, one is created and added to the end of the list (and the test is considered to be a success). If there is only one entry for that input pattern and the entry has an identical output pattern (i.e. the entry is a duplicate), then the test is considered a success and the table is left unchanged. If there is a conflicting entry (one which has an identical input pattern but different output pattern) then the entry is added to the list, but the test is considered a failure. If there is both a conflicting and a duplicate entry, the table is left unchanged and the test is considered to be a failure.

7.3 The Mask Search

The PBIL algorithm is implemented in the MaskSearch class. The operation of MaskSearch is fairly straightforward, as it builds on the capabilities provided by the SystemMask. The MaskSearch class initializes a probability vector for the PBIL algorithm, generates a number of masks from that probability distribution, and tests

these masks against the simulation instance that is currently loaded. The overall function follows the algorithm described in tables 6.2 and 6.3.

Chapter 8

Experimental Results

8.1 Experimental Procedure

Experiments were performed according to the procedure outlined here. The parameters that defined the size of the neighborhood for a cellular automaton and the number of members of that neighborhood were specified, and a number of simulation trajectories for that cellular automaton were generated using the process described in section 4.6. The experimental data was generated using the procedure in table 8.1. Using this data, the PBIL mask search algorithm was run according to the procedure outlined in table 8.2.

8.2 Data Collected

For each set of parameters tested, the number of generations required for each of the n PBIL runs is recorded. From this data, we calculate the sample mean $\bar{X}_n$, sample standard deviation S_n, and quartile statistics for the number of generations required before the search algorithm terminated. The search algorithm terminated if the correct functional mask (a functional mask that identified the exact cells used in the neighborhood function that generated the system) or an equivalent mask (a mask that, while not the neighborhood used to generate the system, is equally or less

Table 8.1: Process for Generating Experimental Data

To test how the search algorithm performed when searching for a mask of a given size, the following procedure was used to generate simulation data to use.

1. The neighborhood parameters are specified. This includes the maximum spatial radius r, the maximum temporal depth d, and the number of cells that are actually members of the neighborhood m.

2. A number of different test samples for these neighborhood parameters are generated.

    (a) A neighborhood N is randomly generated from the parameters specified, and a randomly generated transition function F is created.

    (b) A cellular automaton with 200 cells is generated using the neighborhood N and transition function F.

    (c) 100 different simulation trajectories for this cellular automaton are generated.

        i. The state of each cell in the automaton is randomly initialized.

        ii. The cellular automaton is simulated for 100 time steps, with each state recorded for later analysis.

Using this procedure, data was generated that allows for determining, in general, how well the search algorithm performs on a neighborhood generated using the parameters r, d, and m.

Table 8.2: Process for PBIL Mask Search

Using the experimental data generated according to table 8.1, the following procedure was used to test the performance of the PBIL mask search algorithm. The procedure is performed for each test sample generated for a given set of neighborhood parameters.

1. The PBIL algorithm is initialized with the following parameters:

   - Positive learning rate $\alpha = 0.01$
   - Negative learning rate $\beta = 0.05$
   - Mutation rate $\delta = 0.005$
   - Mutation shift $\gamma = 0.05$
   - Child count $N = 20$
   - Evaluations per child $E = 5000$
   - Initial probability $P_{init} = 0.5$
   - Penalty learning rate $\phi = 0.01$

2. The PBIL algorithm is used to find the original (or an equivalent) mask according to the procedure in table 6.2. During the evaluation of the candidate solutions generated by the PBIL algorithm, E different test cases are selected, and each potential mask is evaluated over the same set of E tests.

   (a) The original mask is defined as being equal to the neighborhood used to generate the cellular automaton. An equivalent mask is one that is not the original mask, but has as few or fewer members than the original mask and is equally able to deterministically represent the functional behavior of the system. For example, if the original mask expressed the system behavior as $a_i^{(t+1)} = F(a_1^{(t)}, a_2^{(t)}, a_2^{(t-1)}, a_3^{(t-1)})$, but the definition of the transition function is such that, for all of the available data from that system, $F(a_1^{(t)}, a_2^{(t)}, a_2^{(t-1)}, a_3^{(t-1)}) = F(a_1^{(t)}, a_2^{(t)}, a_2^{(t-1)})$ then it would be possible (and preferable) for the PBIL algorithm to identify the mask as $M = a_1^{(t)} \times a_2^{(t)} \times a_2^{(t-1)}$ instead of the original.

3. The value of the probability vector P as well as the best mask seen thus far is recorded at each generation for later analysis.

4. Depending on the test, the PBIL algorithm is allowed to run until a mask is found or until some maximum number of generations have been tested.

Table 8.3: Results for $r = 4$, $d = 4$, $m = 4$

Generations Required	(127 tests run)
Minimum	82
25th Percentile (Q_1)	203.75
50th Percentile (Q_2, median)	356
75th Percentile (Q_3)	4,304
Maximum	1,055,752
Sample Mean	22,942.86
Sample Standard Deviation	104,804.44
Actual Mean (95% confidence interval using t-distribution)	$22,942.86 \pm 584.33$

complex than the correct mask and equally valid for defining the functional behavior of the system given all of the available data). Using the sample mean and standard deviation, we also can calculate a 95% confidence interval based on the Student's t-distribution. The true mean μ number of generations required by the PBIL mask search algorithm for the set of parameters being tested has a 95% probability of being within this interval.

8.3 Experimental Results

8.3.1 $r = 4$, $d = 4$, $m = 4$

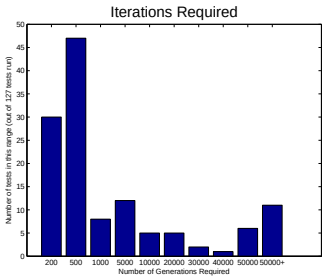
To test how well the mask search procedure performed on these parameters, 127 different tests were performed according to the procedures in tables 8.1 and 8.2. Each test was run until it was able to find the correct mask or an equivalent mask. Of the 127 different tests run, 124 were able to find the correct mask and 3 found an equivalent mask. The results are summarized in table 8.3 and figure 8.1.

8.3.2 $r = 4$, $d = 4$, $m = 8$

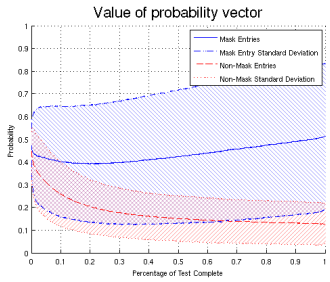
To test how well the mask search procedure performed on these parameters, 150 different tests were performed according to the procedures in tables 8.1 and 8.2. Each test was run until it was able to find the correct mask or an equivalent mask.

Figure 8.1: Results for $r = 4$, $d = 4$, $m = 4$

(a) Generations Required



(b) Probability Vector



(c) Entries in Best Mask

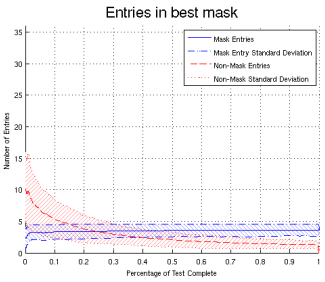


Table 8.4: Results for $r = 4$, $d = 4$, $m = 8$

Generations Required	(150 tests run)
Minimum	155
25th Percentile (Q_1)	645
50th Percentile (Q_2, median)	1,366
75th Percentile (Q_3)	5,421
Maximum	300,235
Sample Mean	10,110
Sample Standard Deviation	31,139
Actual Mean (95% confidence interval using t-distribution)	$10,110 \pm 160$

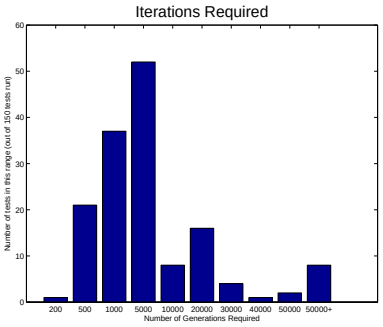
Of the 150 different tests run, all were able to find the correct mask. The results are summarized in table 8.4 and figure 8.2.

8.3.3 $r = 6$, $d = 6$, $m = 4$

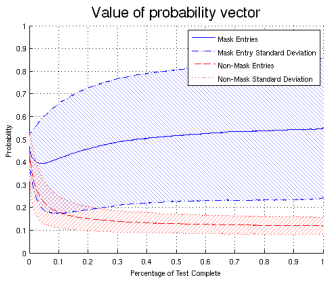
To test how well the mask search procedure performed on these parameters, 150 different tests were performed according to the procedures in tables 8.1 and 8.2. Of the tests run, 97 of these tests were able to find the correct mask in fewer than 100000 iterations; 12 tests were run for more than 100000 iterations until the correct mask was found, and 41 tests were only run for 100000 iterations. The results are summarized in table 8.5 and figure 8.3.

Figure 8.2: Results for $r = 4$, $d = 4$, $m = 8$

(a) Generations Required



(b) Probability Vector



(c) Entries in Best Mask

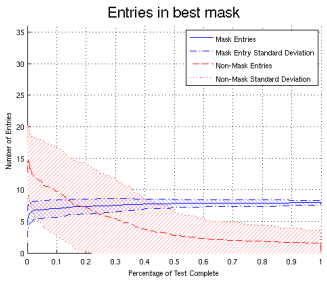


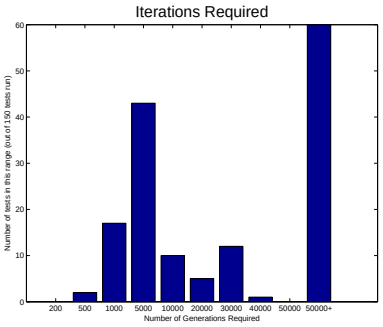
Table 8.5: Results for $r = 6$, $d = 6$, $m = 4$

Generations Required	(109 tests run until completion)
Minimum	441
25th Percentile (Q_1)	1,112.5
50th Percentile (Q_2, median)	2,466
75th Percentile (Q_3)	24,589.25
Maximum	1,410,319
Sample Mean	79,848.39
Sample Standard Deviation	236,719.59
Actual Mean (95% confidence interval using t-distribution)	$79,848 \pm 1,425$

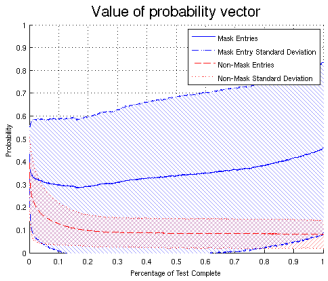
It should be noted that, of 150 tests actually performed, only 109 were completed within a reasonable amount of time. Therefore the statistics reported here are likely somewhat biased to be smaller than the actual values.

Figure 8.3: Results for $r = 6$, $d = 6$, $m = 4$

(a) Generations Required



(b) Probability Vector



(c) Entries in Best Mask

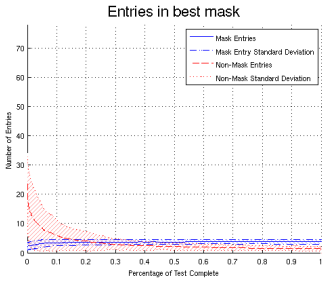
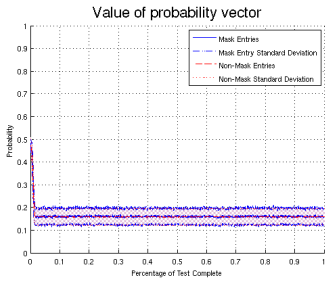
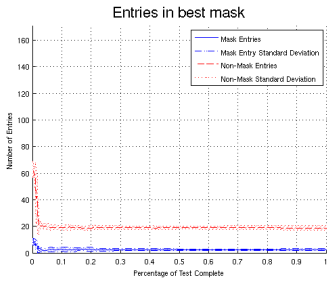


Figure 8.4: Results for $r = 9$, $d = 9$, $m = 19$

(a) Probability Vector



(b) Entries in Best Mask



8.3.4 $r = 9$, $d = 9$, $m = 19$

To illustrate the performance of the algorithm when dealing with large systems (as previously discussed in section 6.2.4), 12 tests were performed according to the procedures in tables 8.1 and 8.2. Each test was run for 100000 iterations and then examined. During these runs, no tests were able to determine the correct mask. Looking at the graphs in figure 8.4, one can see that the probability for the actual mask members follows the same distribution as the probability of non-mask members. From these results, we can infer that it is unlikely that the algorithm would ever find the correct mask.

8.4 Analysis of Results

From the experimental results presented in section 8.3, we can see that the algorithm effectively infers the functional mask that governs the behavior of cellular automata in many situations. As described in section 6.2.4, the algorithm is not very effective when dealing with large potential neighborhoods with relatively few actual members (which can be seen from the results in section 8.3.4).

From the other tests, we can see that, in situations where the algorithm is able to find the correct mask, it does so very efficiently. For the tests run with the parameters $r = 4$ and $d = 4$, we calculated earlier in section 5.2 that there are

$N = 68,719,476,735$ different masks possible. Each test consists of 100 different samples, with each sample consisting of a system with 200 elements being run for 100 time steps. This provides a total of $T = 100 \times 200 \times (100 - 4) = 1,920,000$ different examples of input/output mappings to use. Assuming that we performed a brute force search by randomly generating a candidate mask and testing it over all possible tests, we would have a $\frac{1}{N}$ chance of generating the correct mask on the first try, a $\frac{1}{N-1}$ chance on the second try, a $\frac{1}{N-2}$ chance on the third try, etc... We can calculate the expected number of masks that would have to be tested to be

$$E[N_M] = \sum_{i=1}^{N} \left[i * \prod_{j=1}^{i-1} \left(1 - \frac{1}{N-j} \right) * \frac{1}{N-i+1} \right] \tag{8.1}$$

This quantity was calculated and found to be $E[N_M] = 34,359,738,373 \approx \frac{N}{2}$. So, on average using a brute force method, we can expect to have to test just over half of the possible masks before the optimal one is found. Considering each full-length test of a mask involves $T = 1,920,000$ evaluations, we expect to need $E[N_M]T = 6.60 \times 10^{16}$ actual evaluations before finding the optimal mask.

Using the algorithm presented in this project, we found the optimal mask for $\{r = 4, d = 4, m = 8\}$ in $300,235$ generations and needed $1,055,752$ generations for $\{r = 4, d = 4, m = 4\}$ (for the worst-case tests). Considering each generation involves creating 20 masks and performing $5,000$ evaluations for each one, $30,023,500,000$ evaluations were performed for $\{r = 4, d = 4, m = 8\}$ and $105,575,200,000$ evaluations were needed for $\{r = 4, d = 4, m = 4\}$.

Computing the speedup as

$$S = \frac{T_B}{T_E} \tag{8.2}$$

where T_B is the number of evaluations needed for a brute force search, and T_E is the number of evaluations needed for our algorithm, we find that, in the worst performing test run for $\{r = 4, d = 4, m = 8\}$ and $\{r = 4, d = 4, m = 4\}$, we have a speedup of 2.20×10^6 and 6.3×10^5, respectively.

Chapter 9

Conclusions

In this project, we created a very general implementation of a cellular automaton as a discrete event system simulation and developed and implemented an algorithm for discovering the neighborhood function of a cellular automaton given only simulated trajectories of the system. A number of tests were run by generating simulation trajectories and using the data to infer the functional mask with the population-based incremental learning algorithm.

From the analysis in section 8.4, one can see that the algorithm presented has some promise; in the situations where it is able to find the correct functional mask that it does so much more effectively than a brute force search.

There is still room for improvement; the algorithm presented does not perform well on systems that have large possible masks, and the existing implementation of the algorithm could certainly stand to be optimized. The performance of the implementation could also be vastly improved by making it multi-threaded and running it on a massively parallel machine.

Bibliography

[1] Shumeet Baluja. Population based incremental learning: A method for integrating genetic search based function optimization and competitve learning. Technical report, 1994. 2, 26

[2] Angela Castells. *Qualitative Modeling and Simulation of Biomedical Systems using Fuzzy Inductive Reasoning.* PhD thesis, Universitat Politécnica de Catalunya, 1994. 18, 21

[3] Bastien Chopard and Michel Droz. *Cellular Automata Modeling of Physical Systems.* Cambridge University Press, 1998. 1, 7

[4] K. Culik, L. P. Hurd, and S. Yu. Computation theoretic aspects of cellular automata. *Phys. D*, 45(1-3):357–378, 1990. 7

[5] G.B. Ermentrout and L. Edelstein-Keshet. Cellular automata approaches to biological modeling. *Journal of Theoretical Biology*, 160:97–97, 1993. 2

[6] N. Ganguly, B. K. Sikdar, A. Deutsch, G. Canright, and P. P. Chaudhuri. A Survey on Cellular Automata. Technical report, 2003. 1, 7

[7] G.J. Klir. *An approach to general systems theory.* Van Nostrand Reinhold New York, 1969. 1, 18

[8] A.M. Law, W.D. Kelton, and W.D. Kelton. *Simulation modeling and analysis.* McGraw-Hill New York, 1991. 22

[9] D.H. Li and F.E. Cellier. Fuzzy measures in inductive reasoning. In *Proceedings of the 22nd conference on Winter simulation*, pages 527–538. IEEE Press, 1990. 22

[10] O. Martin, A. M. Odlyzko, and S. Wolfram. Algebraic properties of cellular automata. *Communications in Mathematical Physics*, 93(2):219–258, 1984. 7

[11] K. Nagel and M. Schreckenberg. A cellular automaton model for freeway traffic. *J. Phys. I France*, 2(12):2221–2229, 1992. 2

[12] James Nutaro. adevs: A discrete event system simulator (version 2.4) [software]. Available from: http://www.ornl.gov/∼1qn/adevs/index.html, Accessed 2010. 2, 12

[13] P.K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, 1990. 34

[14] Stephen Wolfram. Universality and complexity in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1-2):1 – 35, 1984. 7

[15] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002. 1, 7

[16] Bernard P. Zeigler and Sankait Vahie. Devs formalism and methodology: Unity of conception/diversity of application. In *In Proceedings of the 25th Winter Simulation Conference*, pages 573–579. ACM Press, 1993. 2, 4