THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

JOHN CONWAY'S GAME OF LIFE IMPLEMENTED AS A NEURAL NETWORK

By

AARON SHUMAKER

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the requirements for Honors in the Major

Degree Awarded:
Summer Semester, 2006

Table of Contents

List of tables

1 John Conway's Game of Life

John Conway is a mathematician who conceptualized a mathematical game that is known as the Game of Life (no relation to the Hasbro board game) or simply Life. Life is a type of system known as a cellular automaton, and it is classified as a 2-dimensional binary outer totalistic cellular automaton (Wolfram, 2002). Life consists of a rectangular array of equally sized squares that change state as time passes. Each square in the game is called a *cell*. Each cell is binary; it may occupy one of two states during a particular time step: *alive* or *dead*. Time is discrete in Life. For totalistic cellular automata a *neighborhood* of cells is defined. In Life the *neighbors* of a cell are those cells with which it shares either a corner or side. Thus each cell in the interior of the array has a neighborhood of the immediate eight surrounding cells, which is described as a Moore Neighborhood (see section 2) with a range of $r = 1$.

The rules of Life were published in Scientific America in 1970. Cells change state with each time step, or generation, according to a set of rules. If a living cell has exactly two or three neighbors, then it will remain in the living state. All other living cells transition to the dead state. The rational is that a cell with 0 or 1 living neighbors will starve, and one with 4 to 8 living neighbors will be overcrowded. If a dead cell has exactly three living neighbors, then it is "born" into the living state. Otherwise, a dead cell remains dead (Gardner, 1970).

These rules can be formalized as follows:

1) **The *survival* rule:** If a cell in the alive state has exactly two or three neighbors, then it will remain in the alive state.
2) **The *death* rule:** If a cell in the alive state has 0, 1, 4, 5, 6, 7, or 8 living neighbors, then it will transition to the dead state.
3) **The *birth* rule:** If a dead cell has exactly three neighbors in the alive state, then it transitions to the alive state (Gardner, 1970).

Implicitly a dead cell remains dead if it has more or less than three neighbors in the alive state.

2 Totalistic Cellular Automata

Wolfram (2002) defines a cellular automaton as *totalistic* if the rules are based only on the total number of cells alive in the defined neighborhood, regardless of their relative location in the neighborhood. A cellular automaton is subclassified as an *outer* totalistic automaton if the center cell which is being updated is excluded from the neighborhood, but the state of the center cell is still considered by the rules (Wolfram, 2002). “In an outer-totalistic cellular automaton, both the center cell value . . . and the outer total . . . are considered” (Weisstein, 2006). I.e., the rules may distinguish between the case of the center cell being alive or dead for a particular outer total. Life is outer totalistic because the rules are based on a total of alive cells that are in the neighborhood, as well as the state of the cell itself that is being updated. As an example, consider the case of Life when there are two living neighbors. If a cell being updated in Life is already alive, then it will remain alive as per the survival rule, otherwise if the cell is already dead, then no rule updates the cell, and it will implicitly remain dead.

Wolfram (2002) often defines totalistic cellular automata as having a neighborhood with range of some integer r. Such a neighborhood is called a Moore Neighborhood (Weisstein, 2006). A *Moore Neighborhood* is a square of cells which is an odd number of cells wide and high (being odd allows the neighborhood to center perfectly over the center cell). The width and height are each $2r+1$. If the cellular automaton is outer totalistic, then the center cell is excluded from the Moore Neighborhood, and otherwise if not outer, then the center cell is included. Life is outer totalistic with a Moore Neighborhood of r=1.

3 Neural Networks

Hertz, Krogh, & Palmer describe an artificial neural network as an interconnected collection of nodes, each of which is a simplified model of an organic neuron. An individual node consists of a number of inputs, an internal processing function, and an activation function which produces an output value. The architecture of an artificial neural network consists of specifications of the processing and activation functions for each node together with the connections between nodes. A connection between nodes represents the conveyance of the output of the upstream node to one of the inputs of the downstream node, and is usually weighted with a parameter to represent the strength of the connection (Hertz et al., 1991). Some connections may actually connect a node to itself, a self connection. Some node inputs may be designated to receive external data.

This thesis uses an architecture where all nodes are identical, use a weighted sum internal processing function, and a discrete activation function that sends a signal of one when the weighted sum falls between two thresholds. When the activation function produces a value of one, then the node is said to *fire*, or be in the *firing* state. Otherwise the activation function produces a value of zero and the node is in the *not-firing* state.

Time is discrete for the neural network architecture in this thesis. The state of each node is initialized by external input during the first time step. During all subsequent time steps the internal processing and activation functions are applied to determine the state of each node. The process of applying the weighted sum and activation functions to update every node's state is called *activation* of the network. The nodes in the network are equally spaced in a rectangular array. There is an output connection from each node to the input of every neighboring node. Additionally, each node has a self connection.

4 Purpose of Research

The purpose of this thesis is to prove that Life can be implemented as a neural network, and then describe how other automata of a similar classification can also be converted into neural networks. The cells in Life correspond to nodes in the neural network. Cells in the living state are represented by nodes that are in the firing state, and dead cells are those where the node is in the not-firing state. One generation in Life is represented by one activation of the neural network.

A particular array of cells in Life corresponds to a pattern of node states in a neural network. Each cell in Life corresponds to a node in the neural network. It must be shown that each node in the neural network behaves just as a cell in Life would, thus showing that the correspondence is an isomorphism. When a time step passes, each node in a neural network should update to a state that corresponds to the state that the corresponding cell would be updated to. The state to which the respective cell is updated is dependant on the rules of Life. The state to which the node updates is dependant on the architecture of the neural network. Thus the rules of Life have a corresponding neural network architecture. The initial states of cells in a particular instance of Life correspond to the initial states of nodes in the corresponding neural network. After one generation of Life, the resulting pattern of cell states should be the isomorphism of the resulting node states after one activation of the corresponding neural network.

The process used to develop a neural network for Life can also be applied to the development of neural network models for the class of 2-dimensional binary outer totalistic cellular automata. A step by step process for performing the conversion from automaton to neural network is given, as well as Cellular Automata Computer Aided Design and Conversion

(CACADAC) software which can produce a corresponding neural network (see section 14). The correspondence of a cellular automaton to a neural network has three major components:

1) Cellular automaton rules correspond to the neural network architecture.
2) Initial cell states correspond to the initial node states.
3) A cellular automaton generation corresponds to an activation of the neural network.

5 Model Space

The architecture of the artificial neural network must be specified so that the isomorphism is created between the set of cells in Life and nodes in the network. The rules of Life specify that the updating of a cell is dependant on the states of neighboring cells. A node receives information about the states of its neighbors via the input connections from upstream nodes. The input connections of a particular downstream node from neighboring upstream nodes should be weighted equally so that no neighbor's output can affect the node more than another neighbor. A connection weighting of $w_N = 0.12$ is used for neighbor connections.

To emulate the Birth rule of Life, each node in a not-firing state should update to the firing state only when three upstream nodes are in the firing state. To emulate the Survival rule of Life, each node in the firing state should update to the firing state only if two or three upstream nodes are in the firing state. The self connection is weighted $w_C = .04$ weighting. The self connection conveys information regarding the state of the node to the activation function. The self connection allows the activation function to differentiate between the case where a firing node has two firing neighbors, and the case where a not-firing node has two firing neighbors. The death rule is represented by those cases where a node updates from a firing state to the not-firing state. Life does not apply a rule to dead cells that are not affected by the Birth rule, and therefore remain dead. Therefore, nodes in the not-firing state which have any number of neighbors other than three should remain in the not-firing state.

Considering the cases where nodes should fire, an activation function for each node is defined as: If the sum of the weighted inputs is within the inclusive range [0.28, 0.4], then the node will be updated to the firing state and send a signal of 1. Otherwise, it will be updated to the not-firing state and send a signal of 0.

6 Implementation Rationale

For the neural network corresponding to Life the connection strengths chosen are somewhat arbitrary, but the implementation of the neural network architecture must meet certain requirements. An implementation of the neural network would function with any weightings as long as they are engineered so that the feedback connection is weaker than any neighboring connection, neighboring connections are all equal, the lower inclusive range for activation equals the sum of two node neighbor connections plus the strength of the feedback connection, and the upper inclusive range for activation equals the strength of three neighbor connections plus the strength of the feedback connection.

7 Proof Outline

The proof is constructed mathematically to show that the neural network implementation is an isomorph of Life. Additionally, the developed CACADAC software demonstrates the implementation of the neural network (see section 14). In Life a cell can have one of two states. The cell can also have zero to eight alive neighbors. There are eighteen possible cases of the two possible cell states combined with nine possible totals for alive neighbors (0 to 8 alive neighbors). The input signal strength to a node is formulated as a function of the node's state and the number of upstream nodes that are active. Similar to a cell in Life, the node in a neural network can occupy one of two states, and there can be zero to eight active upstream nodes, for a total of eighteen possible cases that must be proven isomorphic. It is a small enough number that a proof of each case is simple and straight forward. Table 1 shows the cases for a cell in the cellular automaton and the corresponding cases for a node in the neural network. Cell states of alive and dead respectively correspond to node states of firing and not-firing. Columns show the initial states for both the cell and its corresponding node. A column shows the result of applying the Life Rules to the cell in each case and updating the cell to perform one timestep in Life, and another column shows the resulting state of the node after one activation of the neural network. Thus the isomorphism is shown to hold after one timestep in both the cellular automaton and the neural network.

8 Proof

The proof uses a modified formula based on the weighted sum function used by the McCullogh Pitts model of a neuron as described by Hertz et al. (1991). At timestep *t*, the total input strength, *net(t)*, is formulated as:

$$net(t) = \sum_{j} w_{ij} n_j(t)$$

The weighting, w_{ij}, represents the weighting of the connection from node *j* to node *i*. The value of n_j(t) is either 0 or 1, and represents respectively the output signals of the not-firing or firing state of node *j* at time *t* (Hertz et al., 1991).

Let subscript *i* be 0 to represent the center cell, and then the subscript *j* can be enumerated to expand the summation to:

$$net = w_{00}n_0 + w_{01}n_1 + w_{02}n_2 + w_{03}n_3 + w_{04}n_4 + w_{05}n_5 + w_{06}n_6 + w_{07}n_7 + w_{08}n_8$$

Where n_j and *net* represent the node states and total input strength at time step *t*. The center cell output signal is n_0, and n_1 through n_8 represent the eight neighboring node's output signals. Note that w_{00} is the weighting of the center cell feedback connection which is defined as $w_C = .04$ in section 5 above. The weightings of connections from upstream neighbors are w_{01} though w_{08}. The neighbor connections are defined as $w_N = 0.12$ in section 5 above. Making substitutions with w_N and w_C gives:

$$net = w_C n_0 + w_N n_1 + w_N n_2 + w_N n_3 + w_N n_4 + w_N n_5 + w_N n_6 + w_N n_7 + w_N n_8$$

Since there are only two possible values for signals, and all neighbors share the same weighting of w_N, then the formula can be simplified to $net = w_C n_0 + w_N n_A + w_N n_D$ where n_A is the sum of the signals from neighboring active nodes (and since the signal strength is 1, then n_A represents the count of neighboring alive cells) and n_D (D for dead) is the sum of the signals from neighboring inactive nodes. Since inactive nodes produce no signal, then n_D is always 0, and thus that term can be eliminated to leave:

$$net = w_C n_0 + w_N n_A \tag{1}$$

The activation formula $\Theta(z)$ is defined as:

$$\Theta(z) = \begin{cases} 1, & \text{if } \mathrm{L} \le z \le \mathrm{U} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where $\mathrm{L} = 2w_N + w_C$ and $\mathrm{U} = 3w_N + w_C$. The output signal of the center node during the next time step is determined by the activation formula:

$$n_0(t+1) = \Theta(net(t)) \tag{3}$$

For the neural network representing Life, if *net*(*t*) falls within the range [0.28, 0.4], then the center node will fire as per the activation function and be updated to be in an active state during the next time step. All other values of *net*(*t*) will result in an inactive node.

For a particular cell in Life, there may be 0 to 8 living neighbors, and the center cell may be alive or dead. Thus there are 18 possible cases of living neighbors with an alive or dead cell.

Table 1 is a comparison of the 18 possible cases that occur in Life and the neural network that represents Life. The Current Cell column shows the initial state of a cell for a case in Life, and the $n_0(t)$ column shows the isomorphic case of a node in the neural network. When applying the rules of Life for each case, the cell is update to the state in the Resulting State column. When applying the weighted sum and activation functions to the node for each case, the resulting state is specified in the $n_0(t+1)$ column. Calculations for all eighteen cases show that in every case the resulting states of alive and dead in Life are emulated by the node's respective states of active and inactive. As Table 1 shows, the isomorphism defined for cell states to node states holds after one time step passes in both Life and the neural network. Thus the two representations are isomorph.

Table 1: Life Cases

Living neighbors	**Current Cell**	"Life" rule that applies	**Resulting State**	$n_0(t)$	$w_C n_0$	$w_N n_A$	*net*(t)	*net*(t) is within [L, U]	$n_0(t+1)$
0	**Dead**	No change	**Dead**	**Not-Firing**	0	0	0	No	**Not-Firing**
	Alive	Death	**Dead**	**Firing**	0.04	0	0.04	No	**Not-Firing**
1	**Dead**	No change	**Dead**	**Not-Firing**	0	0.12	0.12	No	**Not-Firing**
	Alive	Death	**Dead**	**Firing**	0.04	0.12	0.16	No	**Not-Firing**
2	**Dead**	No change	**Dead**	**Not-Firing**	0	0.24	0.24	No	**Not-Firing**
	Alive	Survival	**Alive**	**Firing**	0.04	0.24	0.28	Yes	**Firing**
3	**Dead**	Birth	**Alive**	**Not-Firing**	0	0.36	0.36	Yes	**Firing**
	Alive	Survival	**Alive**	**Firing**	0.04	0.36	0.40	Yes	**Firing**
4	**Dead**	No change	**Dead**	**Not-Firing**	0	0.48	0.48	No	**Not-Firing**
	Alive	Death	**Dead**	**Firing**	0.04	0.48	0.52	No	**Not-Firing**
5	**Dead**	No change	**Dead**	**Not-Firing**	0	0.60	0.60	No	**Not-Firing**
	Alive	Death	**Dead**	**Firing**	0.04	0.60	0.64	No	**Not-Firing**
6	**Dead**	No change	**Dead**	**Not-Firing**	0	0.72	0.72	No	**Not-Firing**
	Alive	Death	**Dead**	**Firing**	0.04	0.72	0.76	No	**Not-Firing**
7	**Dead**	No change	**Dead**	**Not-Firing**	0	0.84	0.84	No	**Not-Firing**
	Alive	Death	**Dead**	**Firing**	0.04	0.84	0.88	No	**Not-Firing**
8	**Dead**	No change	**Dead**	**Not-Firing**	0	0.96	0.96	No	**Not-Firing**
	Alive	Death	**Dead**	**Firing**	0.04	0.96	1.00	No	**Not-Firing**

9 Converting a Totalistic Cellular Automaton into a Neural Network

A neural network can be developed for any cellular automaton in the class of 2-dimensional binary outer totalistic cellular automata. A step by step process is used to convert 2-dimensional binary outer totalistic cellular automata into neural networks. The process of defining a totalistic cellular automaton in terms of a neural network is broken down into three steps. Each step corresponds to an isomorphism component listed at the end of section 4 above. The three major steps are divided into sections 10 through 12 as follows: 10 Conversion: Neural Network Architecture, 11 Conversion: Neural Network Initialization, and 12 Conversion: Dynamics.

10 Conversion: Neural Network Architecture

10.1 *Define the Neighborhood of Each Node*

A cell's neighborhood defines the relative positions of neighbors to that cell (Wolfram, 2002). Let *CellNeighbors*(*U*) be the set of neighbors of a cell *U*, the cell's corresponding node be *V*, and *NodeNeighbors*(*V*) be the set of neighbors of a node *V*. A cell is in *CellNeighbors*(*U*) if and only if it is the neighbor of cell *U*. A node is in *NodeNeighbors*(*V*) if and only if it corresponds to a cell in *CellNeighbors*(*U*).

10.2 *Define the Cellular Automaton Rule Cases and Results*

For a particular cell, there are 0 to n_T living neighbors, where n_T is the total number of neighbors. Thus there are $(n_T + 1)$ possible cases of living neighbors. If the cellular automaton is outer totalistic, then the center cell is alive or dead, and thus the number of possible cases are doubled to $2(n_T + 1)$. These cases are specified in Table 2 below. The Living Neighbors column has $(n_T + 1)$ rows that are enumerated with the possible values of n_A, the number of alive neighbor cells from 0 to n_T. For each value of n_A there are two rows, one for the case where the center cell is dead, and the second row for the case where the center cell is alive. For each case, the cellular automaton's rules are applied to determine the cell's resulting state after one time step, and the resulting state of Dead or Alive is specified under the Resulting State column for each case. For this thesis, where a value is specific for the cellular automaton being converted, an asterisk (*) is placed in Table 2. The asterisk indicates that during an actual conversion, an actual value or state name is used instead of the asterisk.

Table 2: Cellular Automaton Cases									
Living Neighbors	**Current Cell**	**Resulting State**	$n_0(t)$	$w_C n_0$	$w_N n_A$	$net(t)$	$\Theta(net(t))$ Fires?	$n_0(t+1)$
0	**Dead**	*						
	Alive	*						
…	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**
	…	**…**	**…**	**…**	**…**	**…**	**…**	**…**
n_T	**Dead**	*						
	Alive	*						

10.3 *Determine the Isomorph Node States for Each Case*

For each cell with a state of Dead or Alive, there is a node representing that cell with a respective state of Not-Firing or Firing. Cell states in the Current Cell column are represented with the appropriate node states in the $n_0(t)$ column. The same mapping of states is made from the Resulting State column to the $n_0(t+1)$ column.

Table 3: Cellular Automaton Cases								
Living Neighbors	**Current Cell**	**Resulting State**	$n_0(t)$	$w_C n_0$	$w_N n_A$	$net(t)$	$\Theta(net(t))$ Fires?	$n_0(t+1)$
0	**Dead**	*	**Not-Firing**					*
	Alive	*	**Firing**					*
…	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**
	…	**…**	**…**	**…**	**…**	**…**	**…**	**…**
n_T	**Dead**	*	**Not-Firing**					*
	Alive	*	**Firing**					*

10.4 *Determine w_N, the Weighting of Connections from Neighboring Upstream Nodes*

The neighbor connection weighting, w_N, is used for all connections from a node to other neighboring nodes. A value for w_N is chosen from the set of real numbers, such that $w_N > 0$. That is not to say that w_N can vary in value throughout the neural network. Once a value of w_N is chosen, then that same value is used for all neighbor connections.

Recall that formula 1 from section 8, $net = w_C n_0 + w_N n_A$, was derived from the case where $8 = n_T$, but is applicable for any n_T since n_A is simply a grouping all terms that represent alive neighbors. The columns $w_C n_0$, $w_N n_A$, and $net(t)$ are preliminary calculations leading up to the definition of $\Theta(net(t))$. For each case in the Table 4, $w_N n_A$ is the product of the chosen w_N and the number of alive neighbors, n_A, for that case.

Table 4: Cellular Automaton Cases

Living Neighbors	Current Cell	Resulting State	$n_0(t)$	$w_C n_0$	$w_N n_A$	$net(t)$	$\Theta(net(t))$ Fires?	$n_0(t+1)$
0	**Dead**	*	**Not-Firing**		$w_N n_A$			*
	Alive	*	**Firing**		$w_N n_A$			*
…	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**
	…	**…**	**…**	**…**	**…**	**…**	**…**	**…**
n_T	**Dead**	*	**Not-Firing**		$w_N n_T$			*
	Alive	*	**Firing**		$w_N n_T$			*

10.5 *Determine w_C, the Weighting of The Node Self Connection*

A value for w_C is chosen from the set of real numbers, such that $w_N > w_C > 0$. The same value of w_C is used for all node self connections. For each case the value of $w_C n_0$ is the product of w_C and n_0, where $0 = n_0$ or $1 = n_0$ for respective node states of Not-Firing and Firing.

Table 5: Cellular Automaton Cases

Living Neighbors	Current Cell	Resulting State	$n_0(t)$	$w_C n_0$	$w_N n_A$	$net(t)$	$\Theta(net(t))$ Fires?	$n_0(t+1)$
0	**Dead**	*	**Not-Firing**	$w_C 0$	$w_N 0$			*
	Alive	*	**Firing**	$w_C 1$	$w_N 0$			*
1	**Dead**	*	**Not-Firing**	$w_C 0$	$w_N 1$			*
	Alive	*	**Firing**	$w_C 1$	$w_N 1$			*
n_A	**Dead**	*	**Not-Firing**	$w_C 0$	$w_N n_A$			*
	Alive	*	**Firing**	$w_C 1$	$w_N n_A$			*
…	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**
	…	**…**	**…**	**…**	**…**	**…**	**…**	**…**
n_T	**Dead**	*	**Not-Firing**	$w_C 0$	$w_N n_T$			*
	Alive	*	**Firing**	$w_C 1$	$w_N n_T$			*

10.6 *Determine the Behavior of the Activation Function*

For each case the value of the weighted sum processing function, *net*(*t*), is the sum of the values of the $w_C n_0$ and $w_N n_A$ columns. The result is a column of unique *net*(*t*) values, I.E. no *net*(*t*) value from one case equals *net*(*t*) for a different case. For each case where the $n_0(t+1)$ column indicates a resulting state of Firing, then the activation function, $\Theta(net(t))$, fires on that value of *net*(*t*). Otherwise, for all other cases where the $n_0(t+1)$ column indicates Not-Firing, then $\Theta(net(t))$ does not fire. The behavior of $\Theta(net(t))$ is not defined for any other value of *net*(*t*), since a properly constructed isomorphic neural network does not produce any other value of *net*(*t*). Note that this definition of $\Theta(net(t))$ is quite different from the more common threshold activation function. Essentially $\Theta(net(t))$ recognizes each case of *net*(*t*) as corresponding to a particular case in the cellular automaton.

Table 6: Cellular Automaton Cases

Living Neighbors	**Current Cell**	**Resulting State**	$n_0(t)$	$w_C n_0$	$w_N n_A$	$net(t)$	$\Theta(net(t))$ Fires?	$n_0(t+1)$
0	**Dead**	*	**Not-Firing**	$w_C 0$	$w_N 0$	$w_C 0 + w_N 0$	*	*
	Alive	*	**Firing**	$w_C 1$	$w_N 0$	$w_C 1 + w_N 0$	*	*
1	**Dead**	*	**Not-Firing**	$w_C 0$	$w_N 1$	$w_C 0 + w_N 1$	*	*
	Alive	*	**Firing**	$w_C 1$	$w_N 1$	$w_C 1 + w_N 1$	*	*
n_A	**Dead**	*	**Not-Firing**	$w_C 0$	$w_N n_A$	$w_C 0 + w_N n_A$	*	*
	Alive	*	**Firing**	$w_C 1$	$w_N n_A$	$w_C 1 + w_N n_A$	*	*
…	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**
	…	**…**	**…**	**…**	**…**	**…**	**…**	**…**
n_T	**Dead**	*	**Not-Firing**	$w_C 0$	$w_N n_T$	$w_C 0 + w_N n_T$	*	*
	Alive	*	**Firing**	$w_C 1$	$w_N n_T$	$w_C 1 + w_N n_T$	*	*

10.7 *Construct the Neural Network Representing the Cellular Automaton Being Converted*

For every cell in the cellular automaton, there is a corresponding node in the neural network. Thus a grid of cells is represented as a grid of neurons. For each node *V* in the neural network, there are input connections from the set of nodes *NodeNeighbors*(*V*) as defined in step

10.1. The weighting of each connection from a neighbor is the value chosen for w_N in step 10.4. For each node, there is a self-connection whose weight is the value w_C chosen in step 10.5. The functions $net(t)$ and $\Theta(net(t))$ perform as defined in step 10.6.

11 Conversion: Neural Network Initialization

Each node V in the neural network is initialized to a state that corresponds to the state of cell U in the cellular automaton. Cell states of alive correspond to nodes states of firing, and cell states of dead correspond to node states of not-firing.

12 Conversion: Dynamics

For every generation of the cellular automaton one activation of the neural network occurs.

13 Generalizing to Non-Outer Totalistic Automata

The conversion process can be generalized to totalistic automata that are not outer totalistic. Since the rules are dependant only on the total number of neighbors, then the Current Cell, $n_0(t)$, and $w_C n_0$ columns of the Table 6 are eliminated. The weighted sum processing function is redefined as $net(t) = w_N n_A$. Table 6 is modified to have $(n_T + 1)$ rows (one row per value of n_A). Nodes only have self connections if each node is included in its own neighborhood. If each node does have a self connection, then the connection is weighted w_N instead of w_C.

14 Computer Aided Conversion

Depending on the number of cells in the neighborhood, assistance of a computer may be helpful in indexing all of the $2(n_T + 1)$ possible cases and calculating their corresponding weighted sum. A spreadsheet or statistical software can be used to simplify enumerating out all the possible cases and calculating weighted sums. With user supplied output values based on the cellular automaton rules for each case, then even the activation function can be defined.

Cellular Automata Computer Aided Design and Conversion (CACADAC) software accompanies this thesis. CACADAC allows a user to define an outer or non-outer totalistic cellular automaton, and then CACADAC displays the connection weightings and activation values, and also simulates the operation of the cellular automaton using a neural network representation. CACADAC usage instructions are specified in Appendix A, and source code is in Appendix B.

References

Gardner, Martin (1970). Mathematical games: the fantastic combinations of John Conway's new solitaire game life. *Scientific American*, 223, 120-123.

Wolfram, Stephen (2002). *A new kind of science*. Champaign: Wolfram Media.

Weisstein, Eric W. Moore neighborhood. *MathWorld--A Wolfram Web Resource*. Retrieved May 23, 2006, from http://mathworld.wolfram.com/MooreNeighborhood.html

Weisstein, Eric W. Outer-totalistic cellular automaton. *MathWorld--A Wolfram Web Resource*. Retrieved May 23, 2006, from http://mathworld.wolfram.com/Outer-TotalisticCellularAutomaton.html

Weisstein, Eric W. Totalistic cellular automaton. *MathWorld--A Wolfram Web Resource*. Retrieved May 23, 2006, from http://mathworld.wolfram.com/TotalisticCellularAutomaton.html

Weisstein, Eric W. Life. *MathWorld--A Wolfram Web Resource*. Retrieved May 23, 2006, from http://mathworld.wolfram.com/Life.html

Author’s Biography

Aaron Shumaker is a student with FSU and works at Science Applications International Corporation (SAIC) in Fort Walton Beach, FL. He is finishing a Bachelor of Science in Computer Science in Summer 2006 via FSU’s Online Distance Learning program. At SAIC Aaron works under friend and manager, Ellen Bergstrom, as a Data Analyst performing error recovery for several contracts supporting databases used by US military medical facilities. He also develops and maintains in-house database applications to support management and tracking of the tasks being performed and data being analyzed, as well as assisting coworkers as-needed with issues regarding databases, connectivity, or computer troubleshooting.

Aaron earned an Associate in Art degree from Chipola Junior College in Marianna, FL with a major in engineering. A required introductory course in C++ programming taught by Nancy Burns was the spark that convinced Aaron that he should pursue a degree in computer science.

While attending high school at the Jackson Academy of Applied Technology (JAAT), principle Randy Free encouraged Aaron to develop his skills with computers. Aaron developed many skills using CAD software, using 2D and 3D graphical art software, developing websites, and working as the school’s sole system administrator during a few months.

Aaron’s family, Cynthia, Byron, and Starr Shumaker, instilled in him a love for nature, art, and music as he grew up in Marianna. He most enjoys playing slide blues on a 12-string guitar, going for walks in the forests and swamps along Dry Creek at his family’s home, and playing online computer games with friends.

Aaron was born in the year 1982 in Houma, Louisiana.

Appendix A: Using CACADAC

CACADAC has been tested on Microsoft Windows XP Professional Edition with version 1.1 of the Microsoft .NET Framework. However, CACADAC should run on any version of Microsoft Windows that has the .NET Framework 1.1 installed.

To start CACADAC, run the CACADAC.exe file. Upon loading, a window will appear with a row of buttons, a vertical slider, and a grid of red and blue squares. The grid of red and blue squares respectively represent alive and dead cells in Life, but are implemented as corresponding firing and not-firing neural network nodes. The user can left click on any node to set its state to firing, and then right click the node to set its state to not-firing. When CACADAC is run, it will by default load a neural network which simulates Conway's cellular automaton Life and initialize a 64 by 64 grid of nodes with randomized initial states. The buttons are used to control the operation of the program, and each button has a specific function:

Start: Click to begin time stepping of the neural network. The Start button will change to a Stop button. The speed at which time steps is controlled by the slider on the left. Move the slider down to step time faster. Move the slider up to step time slower.

Stop: Click to stop time stepping of the neural network. The Stop button will change back to a Start button.

Step: Click to step the neural network forward a specified number of time steps. Specify the number of steps in the box immediately right of the Step button. Care should be taken when entering a large number of steps, as the program will be unresponsive while computations are being performed.

Clear: Click to clear all firing nodes, setting them effectively to the not-firing state.

Save: Click to save the current neural network. A save file dialog is displayed. Use the save file dialog to specify a location and filename for the neural network to be saved to. Files are saved with the extension *.64x64Nodes

Load: Click to load a neural network or states pattern from a previously saved *.64x64Nodes file. Loading the entire neural network will essentially load a simulation of the cellular automaton that was saved to the file previously, as well as the states of the nodes when the neural network was saved.

Automaton – New: Click to define a new cellular automaton. The grid of cells will be cleared, and a white cell displayed in the center, with red cells around it. The white cell represents a center cell, and the red cells represent the neighborhood of that cell. The user should modify the neighborhood of cells around the center cell to specify a new neighborhood. The user defined neighborhood will be used by CACADAC for all nodes in the neural network simulation. Click the Done button to indicate that CACADAC should use the currently drawn neighborhood.

Once the Done button is clicked, a dialog will be displayed that requests the user to specify if their cellular automaton is outer totalistic. If No is chosen, then another dialog is displayed asking whether the center cell is included in its own neighborhood (since the center cell appears only white when drawing the neighborhood).

The StatesOutput dialog is displayed listing each possible case for the cellular automaton defined. The user should apply the rules of the cellular automaton for each case, and place a check in the "Results in life?" checkbox if the case generates life, and leave the checkbox blank if the case for the rule results in death. The weighting of neighbor node connections is displayed in a box at the top of the window. The weighting of each self connection is also displayed, or 0 is displayed if there are no self connections. The Activation Values box indicates a list of values separated by semicolons (;). The list of activation values are those values of *net*(*t*) that the node should fire on. Once the results of each case have been defined, click the **Done** button. A neural network is generated internally to simulate the defined cellular automaton.

Automaton – Current: Displays the current neighborhood as well as the current rules configuration in the StatesOutput dialog. The user may modify the rule results to create a new cellular automaton with the same neighborhood. Click the Done button on the StatesOutput dialog to confirm changes to the rule results.

Appendix B: CACADAC Source Code

CACADAC is written in the C# programming language, and compilation was tested with the Microsoft Visual C# .NET IDE using version 1.1 of the Microsoft .NET Framework. CACADAC consists of the four source code files Modeling.cs, UIForm.cs, StatesOutput.cs, and StateRule.cs containing the following source code:

```
//Begin file Modeling.cs
using System;
using System.IO;
using System.Threading;
using System.Collections;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace CellLifeGame1
{
   [Serializable]
   class SaveFile
   {
      internal ArrayList nodes;
      internal ArrayList activationValues;

      public SaveFile(ArrayList nodes, ArrayList activationValues)
      {
         this.nodes = nodes;
         this.activationValues = activationValues;
      }

   }

   [Serializable]
   public class Position:IComparable
   {
      //position on an X/Y plane
      public int x;
      public int y;
```

```
    public Position()
    {}

    public Position(int  x, int y)
    {
       this.x = x;
       this.y = y;
    }

    #region  IComparable Members

    ///   <summary>
    ///   Performs lexicographical comparison of a position, with
    ///   Position.x being the primary comparison, and Position.y
    ///   the secondary comparison.
    ///   </summary>
    ///   <param name="obj"><c>Position</c> to compare this instance to.
    ///   </param>
    ///   <returns>Returns less than 0 if  this instance is less than
    ///   <c>obj</c>, greater than 0 for greater than, and 0 for equality.
    ///   </returns>
    public int CompareTo(object   obj)
    {
       if(obj is Position)
       {
          Position position =  (Position) obj;
          int   xResult  = this.x.CompareTo(position.x);
          if(xResult!=0)
          {
             return xResult;
          }
          //else x is equal, thus we use y comparison
          return this.y.CompareTo(position.y);
       }
       throw new ArgumentException("object is not a Position");
    }

    #endregion
 }

 [Serializable]
 public class Link
 {
    public Node Destin;//Destination, where   the   link points to

    //an optional weight for use in applications such as neural nets
    private  double weight;

    public double Weight
    {
       get
       {
          return this.weight;
       }
       set
       {
          this.weight = value;
```

```
      }
   }

   public Link(Node node, double initWeight)
   {
      this.Weight = initWeight;
      this.Destin = node;
   }

   public Link(Node node):this(node, 1)   {}

}

[Serializable]
public class Node:IComparable
{

   public double inputSum;

   //on or  off, firing or not-firing
   private  bool active;
   public bool IsActive
   {
      get
      {
         return active;
      }
      set
      {
         active = value;
      }
   }

   //position of node on an X/Y plane
   public Position   position;

   //list of nodes   to which this one connects
   public ArrayList links;

   ///   <summary>
   ///   Creates  a link from <c>this</c> to <c>node</c>.
   ///   </summary>
   ///   <param name="node">The <c>Node</c> to link to.</param>
   public void AddLink(Link link)
   {
      if(links == null)
      {
         links =  new   ArrayList();
      }

      links.Add(link);
   }

   public Node(int   positionX, int positionY)
   {
      position = new Position();
      this.position.x   = positionX;
      this.position.y   = positionY;
      inputSum = 0;
   }

   #region  IComparable Members
   ///   <summary>
   ///   Performs lexigraphical comparison based on <c>position</c>.
   ///   </summary>
   ///   <param name="obj"><c>Node</c> or <c>position</c>
   ///   to compare this instance to.</param>
   ///   <returns>Returns result of <c>position</c> comparison.</returns>
   public int CompareTo(object   obj)
   {

      if(obj is Node)
      {
         Node node = (Node) obj;
         return this.position.CompareTo(node.position);
      }
      else//let Position try to compare the type
      {
         try
         {
            return this.position.CompareTo(obj);
         }
         catch(ArgumentException e)
         {
            throw new ArgumentException(e +  "and object is not a Node");
         }
      }
   }
   #endregion

}

///   <summary>
///   Maintains a representation of the game modeled in internal data.
///
///   A Modeling object maintains the data model through its own thread.
///
///   Requests are made from client objects, which are queued, and
///   the Modeling object will process those requests in a loop.
///
///   When the modeling thread is TimeStepping the data model, any
///   changes made to the model are queued to an updates object, which
///   the client can access and consume.
///
///   When this thread initially starts, it will load an initial default
///   model, and then   wait for some sort of signal
///   from the UIForm   which will prompt ModelThread to load a   model by
///   some means, from a file, etc., or respond to user editing of the
///   displayed UI model by updating the internal  model.   At some  point
///   the   user may signal   the   model to begin function of time
///   computations.
///   </summary>
public class Modeling
```

```
{
   ///   <summary>
   ///   A queue  of updates available for the UI  to consume.
   ///   Each item in the queue is itself a collection listing nodes or
   ///   properties of the model that have changed during an iteration of
   ///   modeling computation.
   ///
   ///   The client must take care not to modify objects in the Q, as they
   ///   are references to actual objects in the data model.
   ///   </summary>
   public Queue updates;

   /// <summary>
   /// Minimum time to complete a processing loop, in milliseconds
   /// </summary>
   private int loopMin;

   private  Thread modelingThread;

   #region  Operational State Management
   ///   <summary>
   ///   <c>OpStateFlag</c> defines the possible   states that the
   ///   <c>ModelThread</c> can operate in.  Either <c>TimeStepping</c>
   ///   when the model is being updated through each time step,
   ///   Paused when the <c>ModelThread</c> is blocked awaiting commands,
   ///   or <c>UpdatePaused</c> or <c>UpdateTimeStepping</c> when the
   ///   model is processing a request to update the model recieved while,
   ///   respectively, <c>Paused</c> or <c>TimeStepping</c>.</summary>
   [Flags()]
      public enum OpStateFlag
   {
      Paused = 1, TimeStepping = 2, Updating = 4, Starting = 8,
      Exiting = 16, UpdateTimeStepping = TimeStepping | Updating,
      UpdatePaused = Paused | Updating
   };

   /// <summary>
   /// <c>opState</c> defines the current operational state.
   /// </summary>
   private OpStateFlag opState;
   //locking object since value types cannot be locked.
   private object opStateLock;

   /// <summary>
   /// <c>OpState</c> indicates the current operational state that
   /// <c>Modeling</c> is in.
   /// </summary>
   public OpStateFlag OpState
   {
      get
      {
         lock(opStateLock)
         {
            return opState;
         }
      }
   }
```

```
private void OpStateChange(OpStateFlag newOpState)
{
   lock(opStateLock)
   {
      opState=newOpState;
   }
}

private  Queue requests;

/// <summary>
/// <c>RequestTimeStepping</c> will set <c>Modeling</c> to change
/// <c>OpState</c> at the next most convenient point.
/// </summary>
public void RequestTimeStepping()
{
   lock(requests)
   {
      requests.Enqueue(OpStateFlag.TimeStepping);
      Monitor.Pulse(requests);
   }
}

public void RequestPause()
{
   lock(requests)
   {
      requests.Enqueue(OpStateFlag.Paused);
   }
}

public void RequestExit()
{
   lock(requests)
   {
      requests.Enqueue(OpStateFlag.Exiting);
   }
}

//sets activation state of node at position p
public void RequestUpdate(Position p, bool a)
{
   lock(requests)
   {
      requests.Enqueue(OpStateFlag.Updating);
      requests.Enqueue(p);
      requests.Enqueue(a);
      Monitor.Pulse(requests);
   }
   Console.WriteLine("after pulse");

}

private void ProcessUpdate()
{
   Position p;
```

```
    if(requests.Peek() is Position)
    {
       p = (Position)requests.Dequeue();
    }
    else
    {
       Console.WriteLine("Error, not position");
       return;
    }

    bool a;
    if(! (requests.Peek() is bool) )
    {
       Console.WriteLine("Error, not bool");
       return;
    }
    else
    {
       a = (bool)requests.Dequeue();
    }

    int nodeIndex = nodes.BinarySearch( p );

    if(nodeIndex >=   0)//if found
    {
       Node node = (  (Node) nodes[nodeIndex]  );
       node.IsActive = a;
       updates.Enqueue(node);
       OnUpdate(EventArgs.Empty);//fire event
    }
    else
    {
       Console.WriteLine("\nError, node not found for RequestUpdate\n");
       return;
    }
 }

 private void ProcessRequests()
 {
    lock(requests)
    {
       //while OpState request has not been satisfied, it is possible
       //that while waiting in paused mode, we may be awoken to find
       //a new request or multiple requests may be made before
       //ProcessRequests is called.

       while( requests.Count>0 )
       {
          if( requests.Peek() is OpStateFlag )
          {
             OpStateFlag request = (OpStateFlag)requests.Dequeue();
             switch (request)
             {
                case OpStateFlag.Paused:
                   opState=OpStateFlag.Paused;

                   //if there are still requests remaining, then
```

```
                  //we should set state to pause, but continue
                  //processing requests since there will likely
                  //be requests for TimeStepping or Updating which
                  //would normally cancel a pause.
                  if( requests.Count == 0 )
                  {
                     Monitor.Wait(requests);
                  }
                  break;

               case OpStateFlag.Updating:
                  opState=OpStateFlag.Updating | opState;
                  ProcessUpdate();
                  if(requests.Count==0)
                  {
                     requests.Enqueue(
                        (~OpStateFlag.Updating) & opState );
                     break;
                  }
                  else
                  {
                     opState = ( (~OpStateFlag.Updating) & opState );
                  }
                  break;

               case OpStateFlag.TimeStepping:
                  opState=OpStateFlag.TimeStepping;
                  break;

               case OpStateFlag.Exiting:
                  opState=OpStateFlag.Exiting;
                  break;

               default:
                  Console.WriteLine
                     ("Error, default reached in ProcessRequest");
                  break;
            }
         }
         //add else if statement for other object types in queue
      }
   }
}
#endregion

///   <summary>
///   The model data as a collection of <c>Node</c> objects.
///   </summary>
private ArrayList nodes;

/// <summary>
/// List of values on which nodes should fire.
/// </summary>
private ArrayList activationValues;

public ArrayList ActivationValues
{
   get
   {
      return activationValues;
   }
}

#region Update event fields and methods.
///   <summary>
///   <c>Update</c> is fired when new data has been flipped to public.
///   </summary>
public event EventHandler Update;

protected virtual void OnUpdate(EventArgs e)
{
   if (Update != null)
      Update(this,e);
}
#endregion

///   <summary>
///   Loads square grid model into <c>nodes</c> collection.
///   </summary>
///

///   <summary>
///   Loads a grid of nodes and builds links between adjacent and
///   diagonally adjacent nodes(the nearest 8).  Most applicable for
///   John Conway's "Game of Life."
///   </summary>
///   <param name="initPosition">Minimum (x,y) origin of grid
///   construction</param>
///   <param name="width">Width of grid in x direction.</param>
///   <param name="height">Height   of grid  in y direction</param>
///   <param name="initialActivity">IsActive state to initialize each
///   node at.</param>
private  void LoadNodesGrid(Position initPosition,
   int width, int height, bool initialActivity)
{
   for( int i = initPosition.x; i<width ; ++i)
   {
      for( int j = initPosition.y; j<height ; ++j )
      {
         nodes.Add(new Node(i,j));
      }
   }

   nodes.Sort();//A lexigraphical position   based ordering
   //This call is not necesary   here in  this special case because
   //the above for   loops ensure that the ArrayList  is
   //already sorted in  this way, but is done here to show
   //that one should sort before doing a BinarySearch below

   //get a list of the surounding nodes and link
   foreach  (Node node in nodes)
   {
      for( int i = -1   ; i<=1 ; ++i )
      {
```

```
                for( int j = -1   ; j<=1 ; ++j )
                {
                   if( (i != 0) || (j != 0))
                   {
                      Position position = new Position();
                      if( i + node.position.x == width)
                      {
                         position.x = 0;
                      }
                      else if( i + node.position.x == -1 )
                      {
                         position.x = width-1;
                      }
                      else
                      {
                         position.x = i + node.position.x;
                      }

                      if( j + node.position.y == height)
                      {
                         position.y = 0;
                      }
                      else if( j + node.position.y == -1 )
                      {
                         position.y = height-1;
                      }
                      else
                      {
                         position.y = j + node.position.y;
                      }

                      int adjacentNodeIndex = nodes.BinarySearch( position );

                      if(adjacentNodeIndex >= 0)//if found
                      {
                         node.AddLink( new Link(
                            (Node) nodes[adjacentNodeIndex], .12D ) );
                      }
                      else
                      {
                         Console.WriteLine("Error, adjacent node not found");
                      }
                   }
                }
             }
             node.AddLink( new Link((Node) node, .04D) );
             node.IsActive = initialActivity;//set initial state
          }


#if DEBUG
          foreach (Node node in nodes)
          {
             foreach (Link link in node.links)
             {
                Console.WriteLine("Node at ({0},{1}) linked to ({2},{3})",
                   node.position.x, node.position.y,
                    link.Destin.position.x, link.Destin.position.y);
                }
            }
#endif

        }

        ///   <summary>
        ///   Loads a (0,0) origin grid.
        ///   </summary>
        ///   <param name="width">Width of grid in x direction.</param>
        ///   <param name="height">Height   of grid  in y direction</param>
        ///   <param name="initialActivity">IsActive state to initialize each
        ///   node at.</param>
        private void LoadNodesGrid(int width, int height, bool initialActivity)
        {
           LoadNodesGrid(new Position(0,0), width, height, initialActivity);
        }

        ///   <summary>
        ///   Loads a (0,0) origin grid with random activities for cells.
        ///   </summary>
        ///   <param name="width">Width of grid in x direction.</param>
        ///   <param name="height">Height of grid in y direction</param>
        private  void LoadNodesGrid(int width, int height)
        {
           LoadNodesGrid(new Position(0,0), width, height, true);
        }

        //requests a new NN to simulate a CA
        public void RequestAutomaton(
           ArrayList neighborhood, ArrayList activationValues,
           double neighborWeight, double centerWeight)
        {
           this.activationValues = (ArrayList)activationValues.Clone();
           RedoLinks(neighborhood, neighborWeight, centerWeight);
        }

        /// <summary>
        /// Returns(via update Q) and sets active a list of nodes that
        /// link to the node at <c>pos</c>
        /// </summary>
        public void RequestActivateNeighbors(Position pos)
        {
           int   nodeIndex = nodes.BinarySearch(pos);

           if(nodeIndex >=   0)//if found
           {
              foreach(Node node in nodes)
              {
                 foreach(Link link in node.links)
                 {
                    if( link.Destin.Equals((Node)nodes[nodeIndex]) )
                    {
                       node.IsActive = true;
                       updates.Enqueue(node);
                    }
            }
         }

         OnUpdate(EventArgs.Empty);//fire event
      }
      else
      {
         Console.WriteLine("\nError, node not found!\n");
      }
   }

   public void RedoLinks(ArrayList neighborhood, double neighborWeight,
      double centerWeight)
   {
      foreach(Node node in nodes)
      {
         node.links = new ArrayList();

         if(centerWeight != 0.0D)
         {
            node.AddLink( new Link((Node) node, centerWeight) );
         }

         foreach(Position pos in neighborhood)
         {
            int width = 1 + ( (Node)(nodes[nodes.Count-1]) ).position.x;
            int height = 1 + ( (Node)(nodes[nodes.Count-1]) ).position.y;

            int relX = (node.position.x - pos.x)%width;
            int relY = (node.position.y - pos.y)%height;

            if(relX < 0)
            {
               relX += width;
            }

            if(relY < 0)
            {
               relY += height;
            }

            int   adjacentNodeIndex =  nodes.BinarySearch(
               new Position(relX, relY)
               );

            if(adjacentNodeIndex >= 0)//if found
            {
               node.AddLink( new Link(
                  (Node) nodes[adjacentNodeIndex], neighborWeight ) );
            }
            else
            {
               Console.WriteLine("\nError, adjacent node not found!\n");
            }
         }
      }
   }
```

```
private void CalculateWeightedSum()
{
   foreach (Node node in nodes)
   {
      if(node.IsActive)
      {
         foreach (Link link in node.links)
         {
            link.Destin.inputSum += link.Weight;
         }
      }
   }
}


public void RequestLoopTime(int loopTime)
{
   loopMin = loopTime;
}

private void CalculateActivation()
{
   foreach (Node node in nodes)
   {

      bool deactivate = true;
      foreach(double dble in activationValues)
      {
         //if input sum == activationvalue, with tolerance 0.0001
         if(  Math.Abs(dble - node.inputSum) <= (0.0001D))
         {
            //for efficiency we only change node state
            //if it is not already that state
            if( !node.IsActive )
            {
               node.IsActive = true;
               updates.Enqueue(node);
            }
            deactivate = false;
            break;
         }
      }

      //if we are active, and did not find activationValue in loop
      if(node.IsActive && deactivate)
      {//then deactivate
         node.IsActive = false;
         updates.Enqueue(node);
      }

      //reset input to 0 so that totals can begin accumulating again
      node.inputSum = 0;
   }

}
```

```
private void TimeStep()
{
   this.CalculateWeightedSum();
   this.CalculateActivation();
}

public void ProcessSave(String filename)
{
   using(  FileStream fileStream = new FileStream(filename,
              FileMode.Create, FileAccess.Write, FileShare.None)  )
   {
      IFormatter formatter = new BinaryFormatter();
      formatter.Serialize(
         fileStream, new SaveFile(nodes, activationValues) );
   }
}

public void ProcessLoad(String filename)
{
   using(  FileStream fileStream = new FileStream(filename,
              FileMode.Open, FileAccess.Read, FileShare.Read)  )
   {
      IFormatter formatter = new BinaryFormatter();

      Object loaded = formatter.Deserialize(fileStream);
      if(loaded is ArrayList)
      {
         nodes = (ArrayList)loaded;
      }
      else
      {
         SaveFile loadedDual = (SaveFile)loaded;
         nodes = loadedDual.nodes;
         activationValues = loadedDual.activationValues;
      }
   }

   foreach(Node node in nodes)
   {
      updates.Enqueue(node);
   }
   OnUpdate(EventArgs.Empty);
}

public void RequestSteps(int steps)
{
   this.RequestPause();
   //wait for pause
   while(OpState != OpStateFlag.Paused){}

   bool updatesCleared = false;
   for(int i = 0; i < steps; ++i)
   {
      TimeStep();

      //to prevent emory being overused, we clear updates
      if(updates.Count > 2*nodes.Count)
      {
         updatesCleared = true;
         updates.Clear();
      }
   }

   //if we cleared updates, then we need to add all cells to updates
   if(updatesCleared)
   {
      updates.Clear();
      foreach(Node node in nodes)
      {
         updates.Enqueue(node);
      }
   }

   OnUpdate(EventArgs.Empty);//fire event
}

public Modeling():this(40,40)
{}

public Modeling(int height, int width)
{
   requests = new Queue();
   opStateLock = new object();
   nodes = new ArrayList();
   updates = new Queue();
   loopMin = 100;

   LoadNodesGrid(height,width);

   System.Random randGen =
      new Random( unchecked((int)System.DateTime.Now.Ticks) );

   foreach (Node node in nodes)
   {
      //random activity
      node.IsActive = Convert.ToBoolean(randGen.Next(2));
   }

   //activation values corresponding to Conway's Life
   this.activationValues =
      new ArrayList( new double[3] {0.28D, 0.36D, 0.40D} );

   foreach (Node node in nodes)
   {
      updates.Enqueue(node);
   }

   modelingThread = new Thread(
      new ThreadStart(this.ModelingThreadEntryPoint)
      );

   modelingThread.Name  = "modelingThread";
```

```
         opState = OpStateFlag.Starting;
         RequestPause();

         modelingThread.Start();
      }

      public void ModelingThreadEntryPoint()
      {
         DateTime timed = DateTime.Now;
         DateTime timed2 = DateTime.Now;
         OnUpdate(EventArgs.Empty);

         while(opState != OpStateFlag.Exiting)
         {
            Console.Write("while loop took: ");
            timed2 = DateTime.Now;
            Console.WriteLine("{0}",timed2-timed);
            timed = DateTime.Now;

            ProcessRequests();
#if DEBUG
            timed2 = DateTime.Now;
            Console.WriteLine("{0}",timed2-timed);
            timed = DateTime.Now;
            Console.WriteLine("Entering TimeStep");
#endif
            TimeStep();

#if DEBUG

            timed2 = DateTime.Now;
            Console.WriteLine("{0}",timed2-timed);
            timed = DateTime.Now;

            Console.WriteLine("Entering OnUpdate");
#endif
            OnUpdate(EventArgs.Empty);//fire event

#if DEBUG
            timed2 = DateTime.Now;
            Console.WriteLine("{0}",timed2-timed);
            timed = DateTime.Now;
            Console.WriteLine("Sleeping");
#endif
            //loopMin milliseconds later
            timed2 = timed.AddMilliseconds(loopMin);

            //only if less than loopMin milliseconds have passed do we sleep
            if(DateTime.Now<=timed2)
            {
               Thread.Sleep(timed2-DateTime.Now);
            }

#if DEBUG
            timed2 = DateTime.Now;
            Console.WriteLine("{0}",timed2-timed);
            timed = DateTime.Now;
#endif


         }
      }

      public Node GetNode(int index)
      {
         return (Node)nodes[index];

      }

   }//class
}//namespace
//End file Modeling.cs

//Begin file UIForm.cs
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Threading;


namespace CellLifeGame1
{

   /// <summary>
   /// Summary description for Form1.
   /// </summary>
   public class UIForm : System.Windows.Forms.Form
   {
      public class RectGrid : System.Windows.Forms.Panel
      {
         private void InitializeComponent()
         {}

         protected override void OnPaintBackground(PaintEventArgs pevent)
         {
            //disable background painting by not calling base
         }
      }

      /// <summary>
      /// Programmer Added Declarations
      /// </summary>
      private Modeling modeling;//computation and data model

      /// <summary>
      /// Form Designer Generated Declarations
      /// </summary>
      private RectGrid panel1;
      private StatesOutput crntStatesOutput;
      private Cell[,] cells;
      private System.Windows.Forms.Button StartStop;

      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.Container components = null;

      private int cellSize=8;
      private int cellsWide=64;
      private System.Windows.Forms.Button btnClear;
      private System.Windows.Forms.Button btnSave;
      private System.Windows.Forms.Button btnLoad;
      private System.Windows.Forms.Button btnDone;
      private System.Windows.Forms.Button automaton;
      private System.Windows.Forms.TrackBar sldrSpeed;
      private System.Windows.Forms.NumericUpDown numSteps;
      private System.Windows.Forms.Button btnStep;
      private System.Windows.Forms.Button btnCurrentAuto;
      private System.Windows.Forms.GroupBox groupAutomaton;
      private int cellsHigh=64;

      public UIForm()
      {
         //
         // Required for Windows Form Designer support
         //
         InitializeComponent();

         ProgrammaticInit();
      }

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      protected override void Dispose( bool disposing )
      {
         if( disposing )
         {
            if (components != null)
            {
               components.Dispose();
            }
         }
         base.Dispose( disposing );
      }

      private void ProgrammaticInit()
      {
         SetStyle(ControlStyles.UserPaint, true);
         SetStyle(ControlStyles.AllPaintingInWmPaint, true);
         SetStyle(ControlStyles.DoubleBuffer, true);

         crntStatesOutput = new StatesOutput();

         cells= new Cell[cellsWide, cellsHigh];
         for(int i=0;i<cellsWide;++i)
         {
            for(int j=0;j<cellsHigh;++j)
            {
               cells[i,j] = new Cell();
            }
         }

         //Create thread that maintains the model of the game.
         modeling = new Modeling(cellsWide,cellsHigh);

         //event hookup
         modeling.Update += new EventHandler(modeling_Updated);

      }


      private void modeling_Updated(object sender, EventArgs e)
      {
         Console.WriteLine("{0} updates.",modeling.updates.Count);

         while(modeling.updates.Count>0)
         {
            Node node = (Node)modeling.updates.Dequeue();

            Cell cell = cells[node.position.x, node.position.y];

            if(node.IsActive)
            {
               cell.Color = Color.Crimson;
            }
            else
            {
               cell.Color = Color.DarkBlue;
            }

            cell.X = node.position.x*cellSize;
            cell.Y = node.position.y*cellSize;
            cell.Size = cellSize;
         }

         if(this.StartStop.Text == "Stop")
         {
            modeling.RequestPause();
         }
         this.panel1.Invalidate();
      }


      #region Windows Form Designer generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.panel1 = new CellLifeGame1.UIForm.RectGrid();
         this.StartStop = new System.Windows.Forms.Button();
         this.btnClear = new System.Windows.Forms.Button();
         this.btnSave = new System.Windows.Forms.Button();
         this.btnLoad = new System.Windows.Forms.Button();
         this.automaton = new System.Windows.Forms.Button();
         this.btnDone = new System.Windows.Forms.Button();
         this.sldrSpeed = new System.Windows.Forms.TrackBar();
         this.numSteps = new System.Windows.Forms.NumericUpDown();
         this.btnStep = new System.Windows.Forms.Button();
         this.btnCurrentAuto = new System.Windows.Forms.Button();
         this.groupAutomaton = new System.Windows.Forms.GroupBox();
         ((System.ComponentModel.ISupportInitialize)
            (this.sldrSpeed)).BeginInit();
         ((System.ComponentModel.ISupportInitialize)
            (this.numSteps)).BeginInit();
         this.groupAutomaton.SuspendLayout();
         this.SuspendLayout();
         //
         // panel1
         //
         this.panel1.Location = new System.Drawing.Point(56, 40);
         this.panel1.Name = "panel1";
         this.panel1.Size = new System.Drawing.Size(512, 512);
         this.panel1.TabIndex = 0;
         this.panel1.MouseUp += new
            System.Windows.Forms.MouseEventHandler(this.panel1_MouseUp);
         this.panel1.Paint += new
            System.Windows.Forms.PaintEventHandler(this.panel1_Paint);
         this.panel1.DragLeave += new
            System.EventHandler(this.panel1_DragLeave);
         this.panel1.MouseMove += new
            System.Windows.Forms.MouseEventHandler(this.panel1_MouseMove);
         this.panel1.MouseDown += new
            System.Windows.Forms.MouseEventHandler(this.panel1_MouseDown);
         //
         // StartStop
         //
         this.StartStop.Location = new System.Drawing.Point(72, 8);
         this.StartStop.Name = "StartStop";
         this.StartStop.Size = new System.Drawing.Size(40, 24);
         this.StartStop.TabIndex = 2;
         this.StartStop.Text = "Start";
         this.StartStop.Click += new
            System.EventHandler(this.StartStop_Click);
         //
         // btnClear
         //
         this.btnClear.Location = new System.Drawing.Point(256, 8);
         this.btnClear.Name = "btnClear";
         this.btnClear.Size = new System.Drawing.Size(40, 24);
         this.btnClear.TabIndex = 3;
         this.btnClear.Text = "Clear";
         this.btnClear.Click += new System.EventHandler(this.btnClear_Click);
         //
         // btnSave
         //
         this.btnSave.Location = new System.Drawing.Point(304, 8);
         this.btnSave.Name = "btnSave";
         this.btnSave.Size = new System.Drawing.Size(40, 24);
         this.btnSave.TabIndex = 4;
         this.btnSave.Text = "Save";
         this.btnSave.Click += new System.EventHandler(this.btnSave_Click);
         //
         // btnLoad
         //
         this.btnLoad.Location = new System.Drawing.Point(352, 8);
         this.btnLoad.Name = "btnLoad";
         this.btnLoad.Size = new System.Drawing.Size(40, 24);
         this.btnLoad.TabIndex = 5;
         this.btnLoad.Text = "Load";
         this.btnLoad.Click += new System.EventHandler(this.btnLoad_Click);
         //
         // automaton
         //
         this.automaton.Location = new System.Drawing.Point(4, 16);
         this.automaton.Name = "automaton";
         this.automaton.Size = new System.Drawing.Size(36, 20);
         this.automaton.TabIndex = 6;
         this.automaton.Text = "New";
         this.automaton.Click += new
            System.EventHandler(this.automaton_Click);
         //
         // btnDone
         //
         this.btnDone.Enabled = false;
         this.btnDone.Location = new System.Drawing.Point(108, 16);
         this.btnDone.Name = "btnDone";
         this.btnDone.Size = new System.Drawing.Size(40, 20);
         this.btnDone.TabIndex = 7;
         this.btnDone.Text = "Done";
         this.btnDone.Click += new
            System.EventHandler(this.btnDone_Click);
         //
         // sldrSpeed
         //
         this.sldrSpeed.Location = new System.Drawing.Point(8, 8);
         this.sldrSpeed.Maximum = 1000;
         this.sldrSpeed.Minimum = 10;
         this.sldrSpeed.Name = "sldrSpeed";
         this.sldrSpeed.Orientation =
            System.Windows.Forms.Orientation.Vertical;
         this.sldrSpeed.Size = new System.Drawing.Size(45, 544);
         this.sldrSpeed.TabIndex = 9;
         this.sldrSpeed.TickStyle = System.Windows.Forms.TickStyle.None;
         this.sldrSpeed.Value = 200;
         this.sldrSpeed.ValueChanged += new
            System.EventHandler(this.sldSpeed_Changed);
         //
         // numSteps
         //
         this.numSteps.Location = new System.Drawing.Point(168, 8);
         this.numSteps.Maximum = new System.Decimal(new int[] {
                                                                 5000,
                                                                 0,
                                                                           0,
                                                                           0});
         this.numSteps.Minimum = new System.Decimal(new int[] {
                                                                           1,
                                                                           0,
                                                                           0,
                                                                           0});
         this.numSteps.Name = "numSteps";
         this.numSteps.Size = new System.Drawing.Size(80, 20);
         this.numSteps.TabIndex = 10;
         this.numSteps.Value = new System.Decimal(new int[] {
                                                                         1,
                                                                         0,
                                                                         0,
                                                                         0});
         //
         // btnStep
         //
         this.btnStep.Location = new System.Drawing.Point(120, 8);
         this.btnStep.Name = "btnStep";
         this.btnStep.Size = new System.Drawing.Size(40, 24);
         this.btnStep.TabIndex = 8;
         this.btnStep.Text = "Step";
         this.btnStep.Click += new System.EventHandler(this.btnStep_Click);
         //
         // btnCurrentAuto
         //
         this.btnCurrentAuto.Location = new System.Drawing.Point(48, 16);
         this.btnCurrentAuto.Name = "btnCurrentAuto";
         this.btnCurrentAuto.Size = new System.Drawing.Size(52, 20);
         this.btnCurrentAuto.TabIndex = 11;
         this.btnCurrentAuto.Text = "Current";
         this.btnCurrentAuto.Click += new
            System.EventHandler(this.btnCurrentAuto_Click);
         //
         // groupAutomaton
         //
         this.groupAutomaton.Controls.Add(this.btnDone);
         this.groupAutomaton.Controls.Add(this.btnCurrentAuto);
         this.groupAutomaton.Controls.Add(this.automaton);
         this.groupAutomaton.Location = new System.Drawing.Point(400, 0);
         this.groupAutomaton.Name = "groupAutomaton";
         this.groupAutomaton.Size = new System.Drawing.Size(152, 40);
         this.groupAutomaton.TabIndex = 13;
         this.groupAutomaton.TabStop = false;
         this.groupAutomaton.Text = "Automaton";
         //
         // UIForm
         //
         this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
         this.ClientSize = new System.Drawing.Size(576, 557);
         this.Controls.Add(this.groupAutomaton);
         this.Controls.Add(this.numSteps);
         this.Controls.Add(this.sldrSpeed);
         this.Controls.Add(this.btnStep);
         this.Controls.Add(this.btnLoad);
         this.Controls.Add(this.btnSave);
```

```
         this.Controls.Add(this.btnClear);
         this.Controls.Add(this.StartStop);
         this.Controls.Add(this.panel1);
         this.FormBorderStyle =
            System.Windows.Forms.FormBorderStyle.FixedToolWindow;
         this.Name = "UIForm";
         this.RightToLeft = System.Windows.Forms.RightToLeft.No;
         this.Text =
            "Cellular Automata Computer Aided Design and Conversion
(CACADAC)";
         ((System.ComponentModel.ISupportInitialize)
            (this.sldrSpeed)).EndInit();
         ((System.ComponentModel.ISupportInitialize)
            (this.numSteps)).EndInit();
         this.groupAutomaton.ResumeLayout(false);
         this.ResumeLayout(false);

      }
      #endregion

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
         Application.Run(new UIForm());
         Application.ExitThread();//allow other threads to continue
      }

      private bool tempPause = false;

      private void panel1_MouseDown(
         object sender, System.Windows.Forms.MouseEventArgs e)
      {
         if(this.StartStop.Text == "Stop")
         {
            this.StartStop.Text = "Start";
            tempPause = true;
            modeling.RequestPause();
         }

         if(e.Button == MouseButtons.Left)
         {
            Position position = new Position(e.X/cellSize,e.Y/cellSize);
            modeling.RequestUpdate(position,true);
         }
         else if(e.Button == MouseButtons.Right)
         {
            Position position = new Position(e.X/cellSize,e.Y/cellSize);
            modeling.RequestUpdate(position,false);
         }
      }

      private void panel1_MouseUp(
         object sender, System.Windows.Forms.MouseEventArgs e)
      {
```

```
   if(this.StartStop.Text == "Start" && tempPause==true)
   {
      this.StartStop.Text = "Stop";
      tempPause = false;
      modeling.RequestTimeStepping();
   }

   if(e.Button == MouseButtons.Left)
   {
      Position position = new Position(e.X/cellSize,e.Y/cellSize);
      modeling.RequestUpdate(position,true);
   }
   else if(e.Button == MouseButtons.Right)
   {
      Position position = new Position(e.X/cellSize,e.Y/cellSize);
      modeling.RequestUpdate(position,false);
   }

}

private void panel1_Paint(
   object sender, System.Windows.Forms.PaintEventArgs a)
{
   for(int i=0;i<cellsWide;++i)
   {
      for(int j=0;j<cellsHigh;++j)
      {
         using( Brush brush = new SolidBrush(cells[i,j].Color) )
            a.Graphics.FillRectangle(brush,
               cells[i,j].X,// + panel1.Location.X,
               cells[i,j].Y,// + panel1.Location.Y,
               cells[i,j].Size,cells[i,j].Size);
      }
   }

   if(this.StartStop.Text == "Stop")
   {
      modeling.RequestTimeStepping();
   }
}

private void StartStop_Click(object sender, System.EventArgs e)
{
   if(this.StartStop.Text == "Start")
   {
      modeling.RequestTimeStepping();
      this.StartStop.Text = "Stop";
   }
   else if(this.StartStop.Text == "Stop")
   {
      modeling.RequestPause();
      this.StartStop.Text = "Start";
   }
}

private void panel1_DragLeave(object sender, System.EventArgs e)
{
   Console.WriteLine("dragleave occured");
}

private void panel1_MouseMove(
   object sender, System.Windows.Forms.MouseEventArgs e)
{
   if(e.Button == MouseButtons.Left)
   {
      Position position = new Position(e.X/cellSize,e.Y/cellSize);
      modeling.RequestUpdate(position,true);
   }
   else if(e.Button == MouseButtons.Right)
   {
      Position position = new Position(e.X/cellSize,e.Y/cellSize);
      modeling.RequestUpdate(position,false);
   }
}

private void btnClear_Click(object sender, System.EventArgs e)
{
   modeling.RequestPause();

   foreach(Cell cell in cells)
   {
      modeling.RequestUpdate(
         new Position(cell.X/cellSize,cell.Y/cellSize), false);
      //Console.WriteLine("Position({0},{1})",cell.X,cell.Y);
   }
}

private void btnSave_Click(object sender, System.EventArgs e)
{
   modeling.RequestPause();

   SaveFileDialog save = new SaveFileDialog();
   save.Filter = cellsWide.ToString() + "X" + cellsHigh.ToString() +
      "Nodes Grid|*.64x64Nodes";

   if(save.ShowDialog() == DialogResult.OK)
   {
      modeling.ProcessSave(save.FileName);
   }
   this.StartStop.Text = "Start";
}

private void btnLoad_Click(object sender, System.EventArgs e)
{
   modeling.RequestPause();

   OpenFileDialog open = new OpenFileDialog();
   open.Filter = cellsWide.ToString() + "X" + cellsHigh.ToString() +
      " Nodes Grid|*.64x64Nodes";

   if(open.ShowDialog() == DialogResult.OK)
   {
      modeling.ProcessLoad(open.FileName);
   }
```

```
    this.StartStop.Text = "Start";
}

private void automaton_Click(object sender, System.EventArgs e)
{
    this.btnCurrentAuto.Enabled = false;
    this.StartStop.Enabled = false;
    this.StartStop.Text = "Start";
    this.automaton.Enabled = false;
    this.btnSave.Enabled = false;
    this.btnLoad.Enabled = false;

    modeling.RequestPause();
    panel1.Paint +=new PaintEventHandler(panel1_PaintCenter);
    panel1.Invalidate();

    //this.btnClear_Click(sender,e);
    foreach(Cell cell in cells)
    {
        modeling.RequestUpdate(
            new Position(cell.X/cellSize,cell.Y/cellSize), false);
    }

    modeling.RequestActivateNeighbors(
        new Position(
        cells[cellsWide/2,cellsHigh/2].X/cellSize,
        cells[cellsWide/2,cellsHigh/2].Y/cellSize
        )
        );

    this.btnDone.Enabled = true;
}

private void panel1_PaintCenter(object sender,
    System.Windows.Forms.PaintEventArgs a)
{
    using( Brush brush = new SolidBrush(Color.White) )
        a.Graphics.FillRectangle(brush,
            cells[cellsWide/2,cellsHigh/2].X,
            cells[cellsWide/2,cellsHigh/2].Y,
            cells[cellsWide/2,cellsHigh/2].Size,
            cells[cellsWide/2,cellsHigh/2].Size);
}

private void btnDone_Click(object sender, System.EventArgs e)
{
    this.StartStop.Enabled = true;
    this.btnDone.Enabled = false;
    this.btnSave.Enabled = true;
    this.btnLoad.Enabled = true;

    panel1.Paint -= new PaintEventHandler(panel1_PaintCenter);
    panel1.Invalidate();

    ArrayList neighborhood = new ArrayList();

    foreach(Cell cell in cells)
```

```
      {
         if( (cell.Color == Color.Crimson) &&
            !( (cell.X == cells[cellsWide/2,cellsHigh/2].X) &&
            (cell.Y == cells[cellsWide/2,cellsHigh/2].Y) )
            )
         {
            neighborhood.Add(new Position(
               (cell.X/cellSize) -
               (cells[cellsWide/2,cellsHigh/2].X / cellSize),
               (cell.Y/cellSize) -
               (cells[cellsWide/2,cellsHigh/2].Y / cellSize)
               ));
         }
      }

      this.IgnoreClicks = true;

      crntStatesOutput = new StatesOutput(neighborhood.Count);
      crntStatesOutput.ShowDialog();

      if(crntStatesOutput.ActivationValues != null)
      {
         modeling.RequestAutomaton(
            neighborhood, crntStatesOutput.ActivationValues,
            crntStatesOutput.NeighborWeight,
            crntStatesOutput.CenterWeight
            );
      }

      this.IgnoreClicks = false;
      this.StartStop.Enabled = true;
      this.btnDone.Enabled = false;
      this.btnCurrentAuto.Enabled = true;
      this.automaton.Enabled = true;
      this.Focus();
   }

   private bool ignoreClicks = false;

   private bool IgnoreClicks
   {
      get
      {
         return ignoreClicks;
      }
      set
      {
         if(value == true && ignoreClicks == false)
         {
            this.Enabled = false;
         }
         else if(value == false && ignoreClicks == true)
         {
            this.Enabled = true;
         }
         ignoreClicks = value;
      }
```

```
}

private void sldSpeed_Changed(object sender, System.EventArgs e)
{
   modeling.RequestLoopTime(this.sldrSpeed.Value);
}

private void btnStep_Click(object sender, System.EventArgs e)
{
   this.StartStop.Enabled = false;
   modeling.RequestPause();
   this.StartStop.Text = "Start";
   modeling.RequestSteps((int)numSteps.Value);
   this.StartStop.Enabled = true;
}

private void btnCurrentAuto_Click(object sender, System.EventArgs e)
{
   this.StartStop.Enabled = false;
   this.StartStop.Text = "Start";

   modeling.RequestPause();
   panel1.Paint +=new PaintEventHandler(panel1_PaintCenter);
   panel1.Invalidate();

   foreach(Cell cell in cells)
   {
      modeling.RequestUpdate(
         new Position(cell.X/cellSize,cell.Y/cellSize), false);
   }

   modeling.RequestActivateNeighbors(
      new Position(
      cells[cellsWide/2,cellsHigh/2].X/cellSize,
      cells[cellsWide/2,cellsHigh/2].Y/cellSize
      )
      );

   ArrayList neighborhood = new ArrayList();

   foreach(Cell cell in cells)
   {
      if( (cell.Color == Color.Crimson) &&
         !( (cell.X == cells[cellsWide/2,cellsHigh/2].X) &&
         (cell.Y == cells[cellsWide/2,cellsHigh/2].Y) )
         )
      {
         neighborhood.Add(new Position(
            (cell.X/cellSize)
            - (cells[cellsWide/2,cellsHigh/2].X / cellSize),
            (cell.Y/cellSize)
            - (cells[cellsWide/2,cellsHigh/2].Y / cellSize)
            ));
      }
   }

   this.IgnoreClicks = true;
```

```
        Node node = modeling.GetNode(0);

        double nodeCenterWeight = 0D;
        double nodeNWeight = 0D;
        foreach(Link link in node.links)
        {
           if(   link.Destin.Equals(node) )
           {
              nodeCenterWeight = link.Weight;
           }
           else
           {
              nodeNWeight = link.Weight;

           }
        }

        crntStatesOutput = new StatesOutput(
           neighborhood.Count, nodeNWeight, nodeCenterWeight,
           modeling.ActivationValues);

        crntStatesOutput.ShowDialog();

        if(crntStatesOutput.ActivationValues != null)
        {
           modeling.RequestAutomaton(
              neighborhood, crntStatesOutput.ActivationValues,
              crntStatesOutput.NeighborWeight,
              crntStatesOutput.CenterWeight
              );
        }

        this.IgnoreClicks = false;

        panel1.Paint -= new PaintEventHandler(panel1_PaintCenter);
        panel1.Invalidate();

        this.StartStop.Enabled = true;
        this.btnDone.Enabled = false;

        this.Focus();
     }
  }

  public class Cell
  {
     Panel panel;
     Color color;

     public Cell()
     {
        panel = new Panel();
        panel.Visible = false;
     }

     public int X
```

```
{
   get
   {
      return this.panel.Location.X;
   }
   set
   {
      if(this.panel.Location.X != value)
      {
         this.panel.Location = new Point(value,this.panel.Location.Y);
      }
   }
}

public int Y
{
   get
   {
      return this.panel.Location.Y;
   }
   set
   {
      if(this.panel.Location.Y != value)
      {
         this.panel.Location = new Point(this.panel.Location.X,value);
      }
   }
}

public Color Color
{
   get
   {
      return this.color;
   }
   set
   {
      this.color = value;
   }
}

public Panel Panel
{
   get
   {
      return this.panel;
   }
}

public int Size
{
   get
   {
      return this.panel.Size.Width;
   }
   set
   {
```

```
            if(this.panel.Size.Width != value)
            {
               this.panel.Size = new Size(value,value);
            }
         }
      }
   }
}
//End file UIForm.cs

//Begin file StatesOutput.cs
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace CellLifeGame1
{
   /// <summary>
   /// Summary description for StatesOutput.
   /// </summary>

   public class StatesOutput : System.Windows.Forms.Form
   {
      private ArrayList stateRules;
      double neighborWeight;
      double centerWeight;
      private System.Windows.Forms.Panel panel1;
      private System.Windows.Forms.Button btnDone;
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.Container components = null;
      private System.Windows.Forms.Label labelNWeight;
      private System.Windows.Forms.Label labelCWeight;
      private System.Windows.Forms.TextBox textNWeight;
      private System.Windows.Forms.TextBox textCWeight;
      private System.Windows.Forms.Panel pnlActivationVals;
      private System.Windows.Forms.Label lblActivationValues;
      private System.Windows.Forms.TextBox textActivationValues;

      private ArrayList activationValues;

      public ArrayList ActivationValues
      {
         get
         {
            return activationValues;
         }
      }

      public double NeighborWeight
      {
         get
         {
            return neighborWeight;
   }
   set
   {
      neighborWeight = value;
      textNWeight.Text = value.ToString();
   }
}
public double CenterWeight
{
   get
   {
      return centerWeight;
   }
   set
   {
      centerWeight = value;
      textCWeight.Text = value.ToString();
   }
}

public StatesOutput(int neighborhoodSize,
   double nWeight, double cWeight, ArrayList activationValues)
{
   this.activationValues = activationValues;

   Console.WriteLine(
      "neighborhoodsize in StatesOutput constructor: {0}",
      neighborhoodSize);

   //
   // Required for Windows Form Designer support
   //
   InitializeComponent();

   stateRules = new ArrayList();

   NeighborWeight = nWeight;
   CenterWeight = cWeight;

   bool isOuter = false;
   if(cWeight == .04D)
   {
      isOuter = true;
   }

   for(int i=0; i<=neighborhoodSize; ++i)
   {
      StateRule stateRule = new StateRule(i,isOuter,false);
      stateRules.Add(stateRule);

      foreach(double actVal in activationValues)
      {
         if(
            Math.Abs(actVal - ( ((double)i) * this.NeighborWeight  ))
            <= (0.0001D))
         {
            stateRule.ResultIsAlive = true;
      }
   }

   if(Math.Abs(CenterWeight - .04D  ) <= (0.0001D))
   {
      stateRule = new StateRule(i,isOuter,true);
      stateRules.Add(stateRule);

      foreach(double actVal in activationValues)
      {
         double linksSummation =
            ((double)i) * this.NeighborWeight + this.CenterWeight;

         if( Math.Abs(actVal - linksSummation ) <= (0.00001D))
         {
            stateRule.ResultIsAlive = true;
         }
      }
   }
   Console.WriteLine("i in States add is {0}",i);
   Console.WriteLine(
      "neighborhoodsize in States add: {0}",neighborhoodSize);
}

this.SuspendLayout();

int j=0;
foreach(StateRule stateRule in stateRules)
{
   Console.WriteLine("j in States attributes is {0}",j);
   stateRule.Location = new System.Drawing.Point(0, 32*j);
   stateRule.Name = "stateRule" + 1;
   //stateRule.Size = new System.Drawing.Size(400, 32);
   stateRule.TabIndex = j;
   panel1.Controls.Add(stateRule);
   stateRule.result.CheckedChanged +=
      new EventHandler(outputChange);
   ++j;
}

int calculatedHeight =
   ((StateRule)stateRules[1]).Size.Height*
   (neighborhoodSize+1)*
   (isOuter?2:1);

if(calculatedHeight < 600)
{
   panel1.Size = new Size(416,   calculatedHeight);
}
else
{
   panel1.Size = new Size(416,   600);
}

this.ClientSize = new System.Drawing.Size(
   panel1.Size.Width + panel1.Location.X*2,
   panel1.Size.Height + panel1.Location.Y);
   this.ResumeLayout(false);
}


public StatesOutput(int neighborhoodSize)
{

   Console.WriteLine(
      "neighborhoodsize in StatesOutput constructor: {0}"
      ,neighborhoodSize);
   //
   // Required for Windows Form Designer support
   //
   InitializeComponent();

   stateRules = new ArrayList();

   // Initializes the variables to pass to the MessageBox.Show method.
   string message = "Is your automaton Outer Totalistic?";
   string caption = "Outer";
   MessageBoxButtons buttons = MessageBoxButtons.YesNo;
   DialogResult outerResult;

   // Displays the MessageBox.
   outerResult = MessageBox.Show(this, message, caption, buttons,
      MessageBoxIcon.Question, MessageBoxDefaultButton.Button1);

   bool isOuter;
   DialogResult centerResult;

   if(outerResult == DialogResult.Yes)
   {
      isOuter  = true;
      CenterWeight = .04D;
   }
   else
   {
      isOuter = false;
      message = "Is the center cell part of the neighborhood?";
      caption = "Center Cell";

      centerResult = MessageBox.Show(this, message, caption, buttons,
         MessageBoxIcon.Question, MessageBoxDefaultButton.Button1);

      if(centerResult == DialogResult.Yes)
      {
         CenterWeight = .12D;
         neighborhoodSize += 1;
      }
      else
      {
         CenterWeight = .0D;
      }
   }

   NeighborWeight = .12D;
   for(int i=0; i<=neighborhoodSize; ++i)
   {
      stateRules.Add(new StateRule(i,isOuter,false));
      if(isOuter)
      {
         stateRules.Add(new StateRule(i,isOuter,true));
      }
      Console.WriteLine("i in States add is {0}",i);
      Console.WriteLine("neighborhoodsize in States add: {0}",
         neighborhoodSize);
   }

   this.SuspendLayout();

   int j=0;
   foreach(StateRule stateRule in stateRules)
   {
      Console.WriteLine("j in States attributes is {0}",j);
      stateRule.Location = new System.Drawing.Point(0, 32*j);
      stateRule.Name = "stateRule" + 1;
      //stateRule.Size = new System.Drawing.Size(400, 32);
      stateRule.TabIndex = j;
      panel1.Controls.Add(stateRule);
      stateRule.result.CheckedChanged +=
         new EventHandler(outputChange);
      ++j;
   }

   int calculatedHeight =
      ((StateRule)stateRules[1]).Size.Height*
      (neighborhoodSize+1)*
      (isOuter?2:1);

   if(calculatedHeight < 600)
   {
      panel1.Size = new Size(416,   calculatedHeight);
   }
   else
   {
      panel1.Size = new Size(416,   600);
   }

   this.ClientSize = new System.Drawing.Size(
      panel1.Size.Width + panel1.Location.X*2,
      panel1.Size.Height + panel1.Location.Y);

   this.ResumeLayout(false);
}

public StatesOutput()
{
   int neighborhoodSize = 8;

   Console.WriteLine(
      "neighborhoodsize in StatesOutput constructor: {0}",
      neighborhoodSize);
```

```
//
// Required for Windows Form Designer support
//
InitializeComponent();

stateRules = new ArrayList();

bool isOuter = true;

for(int i=0; i<=neighborhoodSize; ++i)
{
   StateRule stateRule = new StateRule(i,isOuter,false);

   if(i == 3)
   {
      stateRule.ResultIsAlive = true;
   }

   stateRules.Add(stateRule);

   if(isOuter)
   {
      stateRule = new StateRule(i,isOuter,true);
      if(i == 2 || i == 3)
      {
         stateRule.ResultIsAlive = true;
      }
      stateRules.Add(stateRule);


   }
   Console.WriteLine("i in States add is {0}",i);
   Console.WriteLine("neighborhoodsize in States add: {0}",
      neighborhoodSize);
}

if(isOuter)
{
   CenterWeight = .04D;
}
else
{
   CenterWeight = .0D;
}

NeighborWeight = .12D;

this.SuspendLayout();

int j=0;

foreach(StateRule stateRule in stateRules)
{
   Console.WriteLine("j in States attributes is {0}",j);
   stateRule.Location = new System.Drawing.Point(0, 32*j);
   stateRule.Name = "stateRule" + 1;
   stateRule.TabIndex = j;
```

```
      panel1.Controls.Add(stateRule);

      stateRule.result.CheckedChanged +=
         new EventHandler(outputChange);

      ++j;
   }

   int calculatedHeight =
      ((StateRule)stateRules[1]).Size.Height*
      (neighborhoodSize+1)*
      (isOuter?2:1);

   if(calculatedHeight < 600)
   {
      panel1.Size = new Size(416,   calculatedHeight);
   }
   else
   {
      panel1.Size = new Size(416,   600);
   }

   this.ClientSize = new System.Drawing.Size(
      panel1.Size.Width + panel1.Location.X*2,
      panel1.Size.Height + panel1.Location.Y);

   this.ResumeLayout(false);
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
   if( disposing )
   {
      if(components != null)
      {
         components.Dispose();
      }
   }
   base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
   this.panel1 = new System.Windows.Forms.Panel();
   this.btnDone = new System.Windows.Forms.Button();
   this.labelNWeight = new System.Windows.Forms.Label();
   this.labelCWeight = new System.Windows.Forms.Label();
   this.textNWeight = new System.Windows.Forms.TextBox();
   this.textCWeight = new System.Windows.Forms.TextBox();
this.pnlActivationVals = new System.Windows.Forms.Panel();
this.lblActivationValues = new System.Windows.Forms.Label();
this.textActivationValues = new System.Windows.Forms.TextBox();
this.pnlActivationVals.SuspendLayout();
this.SuspendLayout();
//
// panel1
//
this.panel1.AutoScroll = true;
this.panel1.Location = new System.Drawing.Point(0, 64);
this.panel1.Name = "panel1";
this.panel1.Size = new System.Drawing.Size(448, 32);
this.panel1.TabIndex = 0;
//
// btnDone
//
this.btnDone.Location = new System.Drawing.Point(8, 8);
this.btnDone.Name = "btnDone";
this.btnDone.Size = new System.Drawing.Size(56, 24);
this.btnDone.TabIndex = 1;
this.btnDone.Text = "Done";
this.btnDone.Click += new System.EventHandler(this.btnDone_Click);
//
// labelNWeight
//
this.labelNWeight.Location = new System.Drawing.Point(72, 8);
this.labelNWeight.Name = "labelNWeight";
this.labelNWeight.Size = new System.Drawing.Size(120, 16);
this.labelNWeight.TabIndex = 2;
this.labelNWeight.Text = "Neighbor Link Weight: ";
this.labelNWeight.TextAlign =
   System.Drawing.ContentAlignment.TopRight;
//
// labelCWeight
//
this.labelCWeight.Location = new System.Drawing.Point(264, 8);
this.labelCWeight.Name = "labelCWeight";
this.labelCWeight.Size = new System.Drawing.Size(108, 16);
this.labelCWeight.TabIndex = 3;
this.labelCWeight.Text = "Center Link Weight: ";
this.labelCWeight.TextAlign =
   System.Drawing.ContentAlignment.TopRight;
//
// textNWeight
//
this.textNWeight.Location = new System.Drawing.Point(192, 8);
this.textNWeight.Name = "textNWeight";
this.textNWeight.ReadOnly = true;
this.textNWeight.Size = new System.Drawing.Size(64, 20);
this.textNWeight.TabIndex = 4;
this.textNWeight.Text = "";
//
// textCWeight
//
this.textCWeight.Location = new System.Drawing.Point(376, 8);
this.textCWeight.Name = "textCWeight";
this.textCWeight.ReadOnly = true;
```

```
      this.textCWeight.Size = new System.Drawing.Size(64, 20);
      this.textCWeight.TabIndex = 5;
      this.textCWeight.Text = "";
      //
      // pnlActivationVals
      //
      this.pnlActivationVals.AutoScroll = true;
      this.pnlActivationVals.Controls.Add(this.textActivationValues);
      this.pnlActivationVals.Controls.Add(this.lblActivationValues);
      this.pnlActivationVals.Location = new System.Drawing.Point(0, 32);
      this.pnlActivationVals.Name = "pnlActivationVals";
      this.pnlActivationVals.Size = new System.Drawing.Size(448, 32);
      this.pnlActivationVals.TabIndex = 6;
      //
      // lblActivationValues
      //
      this.lblActivationValues.Location = new System.Drawing.Point(8, 8);
      this.lblActivationValues.Name = "lblActivationValues";
      this.lblActivationValues.Size = new System.Drawing.Size(96, 16);
      this.lblActivationValues.TabIndex = 0;
      this.lblActivationValues.Text = "Activation Values:";
      //
      // textActivationValues
      //
      this.textActivationValues.Location =
         new System.Drawing.Point(104, 8);
      this.textActivationValues.Name = "textActivationValues";
      this.textActivationValues.ReadOnly = true;
      this.textActivationValues.Size = new System.Drawing.Size(336, 20);
      this.textActivationValues.TabIndex = 1;
      this.textActivationValues.Text = "";
      //
      // StatesOutput
      //
      this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
      this.ClientSize = new System.Drawing.Size(448, 405);
      this.Controls.Add(this.pnlActivationVals);
      this.Controls.Add(this.textCWeight);
      this.Controls.Add(this.textNWeight);
      this.Controls.Add(this.labelCWeight);
      this.Controls.Add(this.labelNWeight);
      this.Controls.Add(this.btnDone);
      this.Controls.Add(this.panel1);
      this.Name = "StatesOutput";
      this.Text = "StatesOutput";
      this.Load += new System.EventHandler(this.load);
      this.pnlActivationVals.ResumeLayout(false);
      this.ResumeLayout(false);

   }
   #endregion

   private void btnDone_Click(object sender, System.EventArgs e)
   {
      activationValues = new ArrayList();
      foreach(StateRule stateRule in stateRules)
      {
            if(stateRule.ResultIsAlive)
            {
               activationValues.Add(
                  (  ((double)stateRule.NeighborsAlive) *
                  this.NeighborWeight  ) +
                  (stateRule.CenterIsAlive?this.CenterWeight:0.0D)
                  );
            }
         }
         this.Close();
      }

      private void outputChange(object sender, System.EventArgs e)
      {
         textActivationValues.Text = "";
         foreach(StateRule stateRule in stateRules)
         {
            if(stateRule.ResultIsAlive)
            {
               double activationValue =
                  (  ((double)stateRule.NeighborsAlive) *
                  this.NeighborWeight  ) +
                  (stateRule.CenterIsAlive?this.CenterWeight:0.0D);

               textActivationValues.Text +=
                  activationValue.ToString() + ';';
            }
         }
      }

      private void load(object sender, System.EventArgs e)
      {
         textActivationValues.Text = "";
         foreach(StateRule stateRule in stateRules)
         {
            if(stateRule.ResultIsAlive)
            {
               double activationValue =
                  (  ((double)stateRule.NeighborsAlive) *
                  this.NeighborWeight  ) +
                  (stateRule.CenterIsAlive?this.CenterWeight:0.0D);

               textActivationValues.Text += activationValue.ToString() + ';';
            }
         }
      }
   }
}
//End file StatesOutput.cs

//Begin file StateRule.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;
```

```
namespace CellLifeGame1
{
   /// <summary>
   /// A case in a set of rules with a checkbox for the user to
   /// specify whether the result in that case is dead or alive.
   /// </summary>
   public class StateRule : System.Windows.Forms.UserControl
   {
      internal System.Windows.Forms.CheckBox result;

      private System.Windows.Forms.Label lblNeighborsAlive;
      private System.Windows.Forms.Label lblCenterAlive;
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.Container components = null;

      public readonly int NeighborsAlive;
      public readonly bool CenterIsAlive;
      public readonly bool IsOuter;
      //public readonly double ActivationValue;

      public StateRule(int neighborsCnt, bool isOuter, bool centerIsAlive)
      {
         // This call is required by the Windows.Forms Form Designer.
         InitializeComponent();

         this.NeighborsAlive = neighborsCnt;
         this.CenterIsAlive = centerIsAlive;
         this.IsOuter = isOuter;

         lblNeighborsAlive.Text += neighborsCnt;

         if(isOuter == false)
         {
            lblCenterAlive.Visible = false;
         }
         else
         {
            if(centerIsAlive == true)
            {
               lblCenterAlive.Text += "Yes";
            }
            else
            {
               lblCenterAlive.Text += "No";
            }
         }
      }

      public bool ResultIsAlive
      {
         get
         {
            return result.Checked;
         }
         set
         {
            result.Checked = value;
         }
      }

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      protected override void Dispose( bool disposing )
      {
         if( disposing )
         {
            if(components != null)
            {
               components.Dispose();
            }
         }
         base.Dispose( disposing );
      }

      #region Component Designer generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.result = new System.Windows.Forms.CheckBox();
         this.lblNeighborsAlive = new System.Windows.Forms.Label();
         this.lblCenterAlive = new System.Windows.Forms.Label();
         this.SuspendLayout();
         //
         // result
         //
         this.result.CheckAlign =
            System.Drawing.ContentAlignment.MiddleRight;
         this.result.Location = new System.Drawing.Point(256, 8);
         this.result.Name = "result";
         this.result.Size = new System.Drawing.Size(104, 16);
         this.result.TabIndex = 0;
         this.result.Text = "Results in life?";
         this.result.TextAlign =
            System.Drawing.ContentAlignment.MiddleRight;
         //
         // lblNeighborsAlive
         //
         this.lblNeighborsAlive.Location = new System.Drawing.Point(8, 8);
         this.lblNeighborsAlive.Name = "lblNeighborsAlive";
         this.lblNeighborsAlive.Size = new System.Drawing.Size(128, 16);
         this.lblNeighborsAlive.TabIndex = 1;
         this.lblNeighborsAlive.Text = "Neighbors alive: ";
         //
         // lblCenterAlive
         //
         this.lblCenterAlive.Location = new System.Drawing.Point(144, 8);
         this.lblCenterAlive.Name = "lblCenterAlive";
```

```
            this.lblCenterAlive.Size = new System.Drawing.Size(112, 16);
            this.lblCenterAlive.TabIndex = 0;
            this.lblCenterAlive.Text = "Center is alive? ";
            //
            // StateRule
            //
            this.Controls.Add(this.lblCenterAlive);
            this.Controls.Add(this.lblNeighborsAlive);
            this.Controls.Add(this.result);
            this.Name = "StateRule";
            this.Size = new System.Drawing.Size(368, 32);
            this.ResumeLayout(false);

        }
        #endregion
    }
}
//End file StateRule.cs
```

The members of the Committee approve the thesis of Aaron Shumaker defended on June 29th, 2006.

Dr. Chris Lacher
Professor Directing Thesis

Dr. Ken Shaw
Outside Committee Member

Dr. Piyush Kumar
Committee Member