**Lab 1: The Cellular Automata Simulator**

**INTRODUCTION:**

This lab describes the operation of the Cellular Automata Simulator (Cell.jar) and has some exercises for you to do.

In our first run-through with the simulator, we will keep things simple. There are more options and capabilities that we’ll get to later on.

**Running the simulator:**

Set up a folder (called the **base folder**) on your hard drive. The simulator, and all the input and output files for the simulator, will reside in the same folder.

Copy the file Cell.jar (or Cell ) into your base folder, and double-click on it to start.

**Note:** Do NOT run the simulator from the T-drive – it is very slow because of networking, and clogs the network for everyone else. Copy it to your computer, and run it locally.

**Step 1. Set your options**

There are quite a few options, but for today we will mostly just use the default settings.

In particular, for now, every cell has one of two states:

0 = off = white

1 = on = black

**Step 2. Set your initial pattern**

Initially, every cell is off (0, white). By pointing to a cell and left-clicking, you can turn the cell on (1, black).

The **Toggle Value** option determines the new value of a cell after you click on it, and it is initially 1. To turn a cell off, set the Toggle Value to 0 and click on the cell. To turn a cell on, reset the Toggle Value to 1 and click on the cell.

The **Clear** button will turn off all the cells. The **Center** button will center the pattern in the array.

There is a way to read in and write out patterns to a text file – more later.

Also, we can initialize the pattern with a program – more later.

**Step 3. Enter and parse your program.**

We have to tell the simulator how to update each cell. To do this, we will write very short programs that the simulator will run on each cell.

The programs for the simulator are *expressions*, similar to expressions used in MS Excel, or used in your calculator. It is not hard to learn, but there are some subtleties, and we will spend most of this lab working on how to program the simulator.

Write your program in the **Program** text area at the right center of the screen. Alternatively, you write your program into a text file named program.txtin your base folder. Then click on **Read** to read your program into the text area.

After your program is read in, the simulator must parse and compile the program and build the internal structures needed to update each cell. Click on **Parse**.

Hopefully, you will get the message "Program compiled successfully", and you are ready to go! If not, you will be notified of the line number and type of problem (usually a syntax error). Correct the program and try again.

**Note:** There is some other output to **Parse** that is debugging info for me, that you can ignore.

**Step 4. Run the cellular automata.**

Click on S**tep** to advance the pattern in the cellular automata by one generation.

If you get the message: Automata not built, that means that you forgot to parse the program, or that the program was not parsed successfully.

**What the program does:**

Your program is run by every cell at every generation. It calculates the next state of the calling cell, based on the current state of itself and its neighbors.

So the result of the calculation for a two-state automata is a number, either 0 or 1, for each cell. After this calculation, the cell displays its new value, white or black, 0 or 1.

Note that you cannot change the state of a neighboring cell with the program. You can only change the state of the *current* cell, the cell calling the program.

**PROGRAMMING THE CELLULAR AUTOMATA**

**Form of a program:**

Every line of the program has one of the forms:

condition : newstate

or just:

newstate

The condition is an expression that is either true or false.

If it is true, then the cell is updated with the evaluation of newstate and the program halts for the current cell. If it is false, the next line of the program is executed.

If the line has no condition, then the system just updates the cell to the value of newstate.

A line with no condition should be the last line of the program, since no further line is ever executed.

**Note:** If you have had C++ or Java programming, then the program

condition**1** : newstate**1**

condition**2** : newstate**2**

**. . .**

condition**n-1** : newstate**n-1**

newstate**n**

means the same thing as:

**if** (condition**1**) newstate**1**

**else if** (condition**2**) newstate**2**

**...**

**else if** (condition**n-1**) newstate**n-1**

**else** newstate**n**

**Example 1:**

This is an extremely simple rule. It updates every cell to 1. The program is simply:

1

This will turn every cell to 1 at generation 1, and for all future generations.

**Example 2:**

Here C stands for the current state of the cell (0 or 1), and == is a test

for equality. So C==1 is true only when the current cell C has state 1.

C == 0: 1

C == 1: 0

This says that if the current state C is 0, set the new updated state to 1.

But if the current state C is 1, set the new updated state to 0.

**Exercise 1:**

For any given pattern, describe what this program does.

What happens to a pattern when you advance two generations?

**Variables: Neighbors**

The variables C, N, S, E, W will have the current state of the calling cell (center, C), the north neighbor (N), the south neighbor (S), the east neighbor (E), and the west neighbor (W).

The variables NE, NW, SE, SW will have the current state of the northeast neighbor (NE), the northwest neighbor (NW), etc.

**Exercise 2:**

Turn on a few cells and run the program:

NE

Describe the behavior of the cellular automata.

**Wraparound.**

The abstract definition of a cellular automaton has a grid which is infinite in all directions, but in practice we must work with a finite array of cells.

Our simulator *wraps around* in both the horizontal and vertical direction. So for the cell

in the upper left corner, its north neighbor is the cell in the lower left corner, and its west neighbor is the cell in the upper right corner.

**Exercise 3:**

What is the northwest neighbor of the cell in the upper left corner?

**Expressions: Arithmetic**

The simulator understands the usual arithmetic operations:

N+S addition of state of the North neighbor with the state of the South neighbor

N-S subtraction

N\*S multiplication

N/3 quotient, division by three

N%2 remainder, division by two

**Note:**  Arithmetic is extremely limited with just two states, 0 and 1, but it will be useful for multi-state machines – more later.

**Example 3:**

The following program behaves the same as Example 2:

1-C

**Expressions: Comparisons**

These expressions evaluate to True (1) or False (0):

N == S True if N and S are equal (note two equal signs)

N <= S True of N is less than or equal to S

N < S … less than …

N >= S … greater than or equal to …

N > S … greater than …

N != S … not equal to …

A value of True or False is called a **boolean** value. In our simulator, True is the same as 1, and False is the same as 0. For example, the program expressions:

N==1 : 0

and

N : 0

are the same.

**Exercise 4:**

**4a.** Describe in words the following program:

N==1 : 1

C

**4b.** Try it out on a few patterns. What happens?

**4c.** If we change the order of the two lines:

C

N==1 : 1

we get a different behavior. What is it, and why is it different?

**Notes on programming**:

There are a lot of different ways to program that produce the same results. Even at this very simple level of programming, it is important to program with care. It is usually better to be simple and clear than to be clever. (Although clever programs are fun too!) Write your program in a way that is easy for *you* to understand.

Your program should handle all possible situations. If it is possible that none of the conditions are true, you should have a final line without conditions, the “catch-all” rule that applies if none of the previous conditions hold. A common final line is just

C

This says that if none of the previous conditions are true for the current cell, then don’t change the state.

**Avoid fall-through:**

If your program executes every line and still has not determined a value, this condition is called “fall through”, and the simulator sets the state to 0. BUT your program should not rely on this feature, and your programs should be written so that fall-through never occurs.

**Reading and writing programs**

You can write small programs directly into the Program text box, but as your programs get more complex, it is better to write and edit them in the text file. Write your program into the file program.txt in your base folder, and then click on the **Read** button under Program Commands.

If you click on the **Write** button, the simulator will save the current program into the file outprogram.txt in your base folder. Rename the file, because the simulator will overwrite outprogram.txt the next time you save the program.

**Parentheses and precedence.**

How is

N + S \* C

evaluated? Is the addition done first, or the multiplication?

The rules for the order of algebraic operations are called the precedence rules.

The simulator uses the standard precedence rules you learned for algebra:

multiplication/division/remainder before addition/subtraction

arithmetic before comparisons

So in the above expression, S and C are multiplied first, and then you add N.

You can add parentheses to change the order of operations. So in:

(N+S)\*C

first N and S are added, and then you multiply by C.

**Example 4:**

Place a single cell in the center of the array, and run the program

N+(S\*C)

What happens?

Repeat, with a single cell in the center, using the program

(N+S)\*C

Now what happens?

**Note:** If you are in doubt about the precedence in your program, just add parentheses to ensure that the calculations are done in the order you want.

**Expressions: Boolean Operations**

These expressions evaluate to True (1) or False (0):

**…** **and** **…** True if both conditions are true

**…** **or** **…** True if at least one of the two conditions are true (inclusive or)

~ **…**  True if the condition is false

**…** **xor** **…** True if exactly one of the two conditions is true (exclusive or)

**Example 5:**

Turn a cell off if either its North or South neighbor is off.

Otherwise, do not change the state.

A program for this rule:

(N==0) or (S==0): 0

C

**Note:** The precedence rules will do comparisons before boolean operations, so we could drop the parentheses:

N==0 or S==0: 0

C

**Warning:** It is tempting to write

N or S==0: 0

C

but this is **wrong** because or looks at truth values:

The first condition is

Is N true (that is, is N==1) **or** is S==0 true (or both)?

which is ***not*** the same as N==0 or S==0.

**Summary of precedence rules:**

multiplication/division/remainder before addition/subtraction

arithmetic before comparisons

comparisons before boolean operations

and before or

**Exercise 5:**

We want to write a program that does the following:

If the East neighbor is on, then turn the current cell on.

If the West neighbor is on, then turn the current cell on.

But if both E and W neighbors are on, turn the current cell off.

Otherwise, don’t change the current cell.

So if the initial pattern is just a row (where **.** is a white cell and **■** is a black cell):

**. . . ■ . ■ . . .**

then the next generation should be:

**. . ■ ■ . ■ ■ . .**

**5a:** The following program does not work. Why?

E==1: 1

W==1: 1

E==1 and W==1: 0

C

**5b.** Rewrite the program so that it meets the specifications.

**5c.** Starting with a single cell in the center, run this automata for several generations.

on the pattern. What do you observe?

**Exercise 6: Fredkin’s Rule**

The following rule was devised by Ed Fredkin, formerly a professor at MIT. Add together the values of five cells: C and the four neighbors N, S, E, W. If the sum is even, then the new state is 0. If the sum is odd, then the new state is 1.

The following program implements Fredkin’s rule:

(N+S+E+W+C)%2 == 0: 0

(N+S+E+W+C)%2 == 1: 1

**6a.** Explain in words why this program implements Fredkin’s rule. Can you shorten the

program?

**6b.** Try this rule on a small centered initial pattern. What happens at generation 4, 8,16?

What do you expect to happen more generally?

**Exercise 7:**

In each of these problems, we start with a single black cell in the center of the array.

**7a.** Program the cellular automata so the black cell moves west at each step.

**7b.** Program the cellular automata so the starting black cell is the bottom left corner of a larger and larger square. It should produce the following pattern on successive generations:

**. . . . . . . . . . . . ■ ■ ■ ■**

**. . . . . . . . ■ ■ ■ . ■ ■ ■ ■**

**. . . . ■ ■ . . ■ ■ ■ . ■ ■ ■ ■**

**■ . . . ■ ■ . . ■ ■ ■ . ■ ■ ■ ■ …**

**7c.** Program to produce a line that grows up and down in the vertical direction.

It should produce the following patterns on successive generations:

**. . . ■**

**. . ■ ■**

**. ■ ■ ■**

**■ ■ ■ ■ …**

**. ■ ■ ■**

**. . ■ ■**

**. . . ■**

**Variables: Sum Variables**

These are several special variables that the simulator recognizes that implicitly calculates the sum of the neighboring states.

The following variables contain the sum of the values of neighboring cells.

sum4 sum of the four neighbors N, S, E, W

sum5 sum4 plus C

sum8 sum of the eight neighbors: N, NW, W, SW, S, SE, E, NE

sum9 sum8 plus C

Example: For the central cell in the following pattern

**. ■ .**

**. ■ ■**

**■ . ■**

sum4 is 2, sum5 is 3, sum8 is 4, and sum9 is 5.

**Exercise 8:**

Write a program that turns a cell from off to on if exactly one of its eight neighbors is on, and otherwise doesn't change the state. Start with a single black cell in the center and describe what happens.

**Exercise 9:**

Write a program that turns a cell from off to on if exactly two of its eight neighbors are on, and otherwise doesn't change the state. Start with a  block of black squares in the center and describe what happens.

**More variables: local information**

In general, your program does not know the row or column number of the current cell, or what generation it is. We want the rules for cellular automata to be *uniform* in time and space. But there are a few variables that contain non-local information:

BN True (1) if the cell is on the north boundary of the array

BS True (1) if the cell is on the south boundary of the array

BE True (1) if the cell is on the east boundary of the array

BW True (1) if the cell is on the west boundary of the array

The simulator recognizes variables that give the *parity* of the cell position and the *parity* of the generation number:

ROWP 0 if the row number of the current cell is even, 1 if odd.

COLP 0 if the column number of the current cell is even, 1 if odd.

PHASE 0 if the generation number is even, 1 if odd.

The topmost row has row number 0, and the leftmost column has column number 0.

**Exercise 10:**

**10a.** Write a program that turns a cell from off to on if it lies on any of the four *corners*, or if at least *one* of its eight neighbors are on. Otherwise, the cell does not change. Start with an empty pattern. What do you observe?

**10b.**  Write a program that turns a cell from off to on if it lies on any of the four *borders*, or if at least *three* of its eight neighbors are on. Otherwise, the cell does not change. Start with an empty pattern. What do you observe?

**10c**. Write a program that turns a cell from off to on if it lies on any of the four *borders*, or if at least *four* of its eight neighbors are on. Otherwise, the cell does not change. Start with an empty pattern. What do you observe?

**10d.** Start with a single cell in the center of the array. Write a program that moves the cell East, then North, then East, then North in a zigzag pattern. (Hint: use PHASE)

**10e.** Start with a single cell in the NW corner. Write a program that moves the cell East, then South, then West, then North, back to its original position. (Hint: use ROWP, COLP.)

**Random numbers:**

The function ran(a,b) generates a random number 0 or 1, where the ratio of 0’s

to 1’s is the ratio of a to b. For example:

ran(1,1) generates 0's and 1's with approximately equal frequency

ran(50,50) generates 0's and 1's with approximately equal frequency

ran(1,3) generates 25% 0’s and 75% 1’s.

ran(9,1) generates 90% 0's and 10% 1's.

The ran() function can have any number of arguments. The function ran(a,b,c,d), for example, generates random numbers 0, 1, 2, 3 with proportions a:b:c:d. For example

ran(1,1,1,1) generates 0, 1, 2, 3 with equal frequency

ran(1,2,1) generates 0, 1, 2 with frequencies 25% 0’s, 50% 1’s, 25% 2’s.

**Exercise 11:**

What does ran(1,2,3,4) do?

**Initialization Program:**

The program area contains the program called to *update* each cell to its new state. It may also contain a second program, which *initializes* the cellular space. The initialization program (which is optional) comes after the update program and is separated with a line of three or more equal signs: =========

When you click on **Parse**, both the update function and the initialization function are compiled. To run the initialization program and initialize the array, click on **Init**.

**Example 6:**

In the following program, the cellular space is initialized to random 0’s and 1’s with equal frequency, and the update program just moves the pattern to the right:

W

======

ran(1,1)

Enter the program, parse it, and click on **Init** to initialize the pattern. Click on Step to move the pattern. If you click on **Init** again, you get a new random pattern.

**Example 7: Voting Automata**

In this automaton, the states of a cell and its eight neighbors are added up, producing a number from 0 to 9. If the majority of the cells are one, then the center cell is on for the

next generation. If the majority of the cells are zero, then the center cell is off for the next generation. The following program implements this automaton, and initializes the space to random 0’s and 1’s with equal frequency.

sum9 >= 5: 1

sum9 <= 4: 0

==========

ran(1,1)

Try it out!

**Exercise 12: Anneal Automata**

Modify the above program so that if the sum of the nine cells is exactly 4 or greater than 5, then the next generation is on. If the sum of the nine cells is exactly 5 or less than 4, then the next generation is off. Try this automata with an initialization of random 0’s and 1’s with equal frequency. What are the significant differences in the patterns generated by Voting Automata and the Anneal Automata?

**Exercise 13: Shift Down**

Write a program that, whenever a black cell is above a white cell, shifts the black cell down one cell, but does not shift black cells past the south border.

**Note:** This program is a little tricky! How can you use the cell counts displayed at the bottom of the simulator window to check your program?

Try out the program with random 0's and 1's, equally distributed. Eventually the pattern will reach a stable state where it no longer changes from generation to generation. What does the final pattern look like?

**Example 8**: **Soil erosion model**

We build a simple model of soil erosion. In this model, the presence of topsoil is represented by 1, and its absence due to erosion is represented by 0.

A cell with state 1 (representing soil) will erode (change to 0) unless there is soil on the north side (at least one of NW, N, NE is on), and soil on the south side (at least one of SW, S, SE is on), and similarly on the east side, and similarly on the west side.

Here is a program for this model:

C==0: 0

NW+N+NE == 0: 0

SW+S+SE == 0: 0

NW+W+SW == 0: 0

NE+E+SE == 0: 0

1

**Exercise 14: Soil Erosion**

**14a.** Try out the soil erosion automaton with different random initializations: 70% on, 80% on, 90% on. What do you observe?

**14b.** Modify the program so that the cell erodes if a corner instead of a side is eroded. For example, if N,NW,W are all zero, then C becomes zero, and similarly for the other three corners. Try it out on the same initializations: 70%, 80%, 90%. What do you observe?

**Select function:**

The function sel( a, b, c, d, ...) evaluates a, then:

if a is 0 then the function returns b

if a is 1 then the function returns c

if a is 2 then the function returns d, etc.

The sel function is often used with the ran function for the first argument.

Select b, c, d,… according to the frequencies specified by ran.

**Exercise 15:**

Here is a program that randomly replaces a cell with its N neighbor or its E neighbor, with equal probability. Try it out with a random initialization, where white cells are twice as common as black cells. Here is a program:

sel( ran(1,1), N, E)

======

ran(2,1)

What do you observe?

**Exercise 16:**

Describe in words what the following automata does:

sel(ran(1,1,1,1),N,S,E,W)

Initialize the space with random 0’s and 1’s with equal frequency. Do later generations appear as random 0’s and 1’s as well? Describe what you see.

**Exercise 17: Exponential decay**

Program the automaton to start with all white cells.

At each step, any black cell remains black, and any white cell has a 10% chance of turning black. Program this and describe the behavior of the automaton. Try running it a few times and report the average number of generations before it turns all black.

**Exercise 18: Border-Hollow automaton**

The border-hollow rule works as follows:

On even generations (PHASE is 0), a cell is turned on if it is already on, or if any of its eight neighbors are on. Otherwise it does not change.

On odd generations, a cell turns off if all of its eight neighbors are on. Otherwise it does not change.

**18a.** Start with a single black cell in the center and describe what happens in each phase.

**18b**. Try it again, with 5% black cells in a random arrangement.

**Parse and the lookup table**

The simulator maintains an internal lookup table to speed up the process of updating the cells on every generation. When you click on Parse and the simulator reports:

Program compiled successfully

then the simulator has compiled your program and built an internal lookup table. But if the program is complex or uses a lot of variables, there may not be room for the lookup tables, and the simulator reports:

Program compiled successfully – no table.

The simulator will still run your program, although it will run somewhat slower without the lookup table.

**Variables**: **ROW and COL**

For an initialization program (unlike the update program), you can use the variables ROW and COL for the row and column number. For large cells, the rows are numbered 0 (top) to 49 (bottom), and the columns are numbered 0 (left) to 49 (right).

**Example 9:**

The following program starts with a disc of black squares of radius 20, and each generation the cells on the border of the disc turn off.

sum9 <= 5: 0

C

======

(ROW-25)\*(ROW-25) + (COL-25)\*(COL-25) <= 20\*20

**Exercise 19:**

Produce an automata that inverts each cell. Start with a pattern that has every third column on, and all others off. (Hint: use COL and the % operator.)

**More States: Four State and Eight State Automata.**

So far our automata have had just two states. We can set the automata to have four states

by selecting the **4x1** button under Plane Type, or eight states by selecting the **8x1** button. (The second number, 1, 2, or 4, is the number of states in the second cellular plane and will be explained later on.) The color assignments are:

0 white

1 black

2 red

3 green.

The color assignments are listed below the cellular array, as well as the population of each cell type for each generation.

**Note:** Whenever you change the Plane Type, you have to **Parse** the program again.

**Exercise 20:**

Turn on the **4x1** button and run the automata in Exercise 16 using four colors. Initialize the pattern with random cell values 0, 1, 2, 3 in equal proportion. What do you observe?

**Count Variables:**

Our last group of special variables are the *count* variables, which count how many neighbors have a certain value. The variables are:

C4\_0 Number of 4-neighbors (N,S,E,W) equal to 0.

C4\_1 Number of 4-neighbors (N,S,E,W) equal to 1.

C4\_2 similar.

C4\_3

C8\_0 Number of 8-neighbors (N,NE,E,SE,S,SW,W,NW) equal to 0.

C8\_1 Number of 8-neighbors (N,NE,E,SE,S,SW,W,NW) equal to 1.

C8\_2 similar.

C8\_3

The variable C5\_0 counts how many of the five cells (4-neighbors + center cell) are zero, and the variable C9\_0 counts how many of the nine cells (8-neighbors + center cell) are zero. Similarly for C5\_1, C5\_2, C5\_3, C9\_1, C9\_2, C9\_3. Later we will use eight states, and have for example C4\_0 through C4\_7, and similar for the other groups.

For two states, C4\_1 and sum4 are the same, and C8\_1 and sum8 are the same. The count variables are most useful when you have four or eight states.

We conclude this lab with a few programming exercises.

**Exercise 21:**

Program a cellular automata that starts with a single black cell in the center and grows to the right in a triangular shape as follows:

**. . . . . . . . . .**

**. . . . . . ■ . . .**

**. . . ■ . . ■ ■ . .**

**. ■ . ■ ■ . ■ ■ ■ .**

**■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ...**

**. ■ . ■ ■ . ■ ■ ■ .**

**. . . ■ . . ■ ■ . .**

**. . . . . . ■ . . .**

**Exercise 22:**

Using four colors, program the automata so that, starting with a single black cell,

at generation 1 it produces a 2x2 checkerboard of black and red squares,

at generation 2 it produces a 3x3 checkerboard of black and red squares,

at generation 3 it produces a 4x4 checkerboard of black and red squares, etc.

The lower left cell should remain stationary. Write your program so that it still works if you start with a single red square instead of a single black square.

**Exercise 23:**

Program an automaton that, starting with a 2x1 block, produces:

**. . . . . . . . . . ■ .**

**. . . . . . . . . . ■ .**

**. . . . . ■ . . . ■ ■ .**

**. . . . . ■ . . . ■ ■ .**

**■ . . . ■ ■ . . ■ ■ ■ .**

**■ . . . ■ ■ . . ■ ■ ■ .**

generation 0 generation 2 generation 4 **. . .**

The lower left cell should remain stationary.

**Exercise 24:**

This automata was one of the first attempts to simulate a population of living things. It is called the Ulam “dog-bone” automata, named after the inventor of cellular automata, the mathematician Stanislaw Ulam, and the funny shapes it produces.

Set the plane type for four colors, but we will use only three colors:

2: young cell (red)

1: old cell (black)

0: empty, or dead cell (white)

Every young cell turns old in the next generation, and dies the generation after that.

For an empty cell, a young cell is born if exactly one of its four neighbors contains a cell

(young or old). Program this automata, starting with a single young cell in the center.

(Hint: The use of count variables like C4\_0 simplifies the program.)

**Report due** Wednesday night, July 17.

Please email to [mayans@fdu.edu](mailto:mayans@fdu.edu) – one report per group, all names in the report.

Please send me your file as a separate attachment to the email, \*not through Google Docs\*.

See you next time!