Practical sheet #5 Cellular Automata

Before you begin this exercise, you must have prepared by reading through the iBook, *Biological Bits*, section 3.1, pp. 55-60. It will also help you to read the online resource at Wolfram MathWorld which provides a detailed discussion of 1D Cellular Automata.[1]

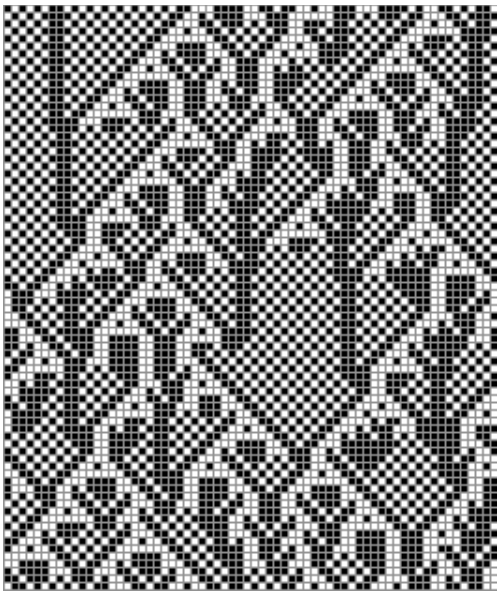

Image © Alice Eldridge

The purpose of this exercise is to build and explore the complex emergent behaviour of 1-bit, 1D Cellular Automata.

Part A : Cellular Automaton Implementation

Implementing the CA class.

1) Within a new class called CA, implement a simple array of Boolean values called *stateArray1*. The array will store a set of variables representing the state of your CA's cells. Specify the length of the array with a const class variable. Also, add a pointer called *currentState* to the class. This should, to begin with, point to the *stateArray1*.

2) Implement a second identical data-structure, *stateArray2*. This too will store the state of the CA's cells. To begin with, it will store the future state of your CA after the rules have been applied to the cells in *stateArray1*. Create a new pointer called *futureState* which, initially, should point to *stateArray2*.

3) Implement a method for swapping the pointers *currentState* and *futureState* between the two arrays *stateArray1* & *2*.

4) Implement an output method for printing to the console, on a single row with a newline character at the end, the current state of the CA. I suggest printing *false* as the character "-" and *true* as the character "X". This looks pretty, but really, use any characters of equal width you please!

5) Implement a method to allow the user to manually enter a series of true and false values (or whatever characters you are using) to initialise the array pointed to by *currentState*. Tell the user the length of the array they need to fill.

6) Implement a method *getLeftNeighbourState()* that takes a parameter which is a valid index into the array indicated by *currentState* and returns the state of the left neighbouring cell. If the *index*=0, treat cell at *index*=*arrayLength*-1 as its left neighbour.

7) Implement a method *getRightNeighbourState()* that takes a parameter which is a valid index into the array pointed to by *currentState* and returns the state of the right neighbouring cell. If the *index*=*arrayLength*-1, treat cell at *index*=0 as its right neighbour.

---

[1] http://mathworld.wolfram.com/ElementaryCellularAutomaton.html

Implementing a CA rule set class.

A CA has a set of transformation rules. A 1-bit rule has a "left" side that is a bit-pattern which, in our case, is a sequence of 3 Boolean values. Its "right" side is a single Boolean value. We can represent it like this (X,Y,Z; A) where X, Y, Z and A are Boolean values.

Each rule in the set is tested against each cell in the current state of the CA as follows:

```
For each cell in the CA's currentState array
    currentCellState      ← getCellState(cell)
    leftNeighbourState    ← getLeftNeighbourState(cell)
    rightNeighbourState   ← getRightNeighbourState(cell)

       For each rule (X,Y,Z; A) in the rule set
         if ((currentCellState=Y) && (leftNeighbourState=X) && (rightNeighbourState=Z))
         {
              setCellFutureState(cell, A)
         }
```

8) Implement a RuleSet class to be used as described above.

It will need a container for a set of rules. For a 1-bit CA, how many rules will you need in a set? How should these rules be ordered?

9) The class will need a method for specifying the rule set to be used. A single string of true and false values (specified with whatever characters you are using) should be entered from the keyboard to fill in the righthand sides for all of your rules. The lefthand side of the rules needn't be specified by the user. Why not?

10) Insert a RuleSet data member into your CA class.

11) Add a method to the CA class to apply its RuleSet to the array pointed to by *currentState*. Compute each cell's future state and enter this into the array pointed to by *futureState*.

Once a rule has been applied to the array pointed to by *currentState*, what do you need to do with the class' array pointers?

12) Now set up a *main()* method with a loop that repeatedly applies the rule set to the CA and prints out the current state of the CA after each iteration.

Part B: Cellular Automaton experimentation.

Initialise your *currentState* array with a length of 41 and a single, central True. Initialise all other cells in the *currentState* to False. You can hardcode this configuration into the CA class' default constructor if you like (rather than forcing the user to enter it manually each time).

Specify and record (e.g. by saving in a plain text file) 5 different rule set specifications alongside their decimal values. Save the output of applying each rule set 20 times beginning with the proscribed initial CA state.

How might the output these rules generate fit within Wolfram's classes of Cellular Automaton behaviour detailed in *Biological Bits* (p.58)?

Compare the output of each rule you tried to those listed in the online table here:
http://mathworld.wolfram.com/ElementaryCellularAutomaton.html

Is your software working correctly?

Test your software by applying Rule 30, i.e., 00011110 (binary) = 30 (decimal) which specifies a rule set like this:

Lefthand side	111	110	101	100	011	010	001	000
Righthand side	0	0	0	1	1	1	1	0

Rule 30 is well-know. Have a read of its Wikipedia page:
http://en.wikipedia.org/wiki/Rule_30

Why is this rule set interesting? Were any of the other rule sets you tried out as interesting as this one? Which ones?