CSE 30151 Theory of Computing
Fall 2017
Project: Cellular Automata

Version 1: Jan. 17, 2018

Contents

1 Overview

You have all probably seen Conway's "Game of Life" [3] The purpose of this project is to introduce you to a generalization of that game, termed **cellular automata** or **CA**. In this paradigm there is a possibly infinite grid of *cells* which interact with each other via a set of rules that use some region around each cell. Depending on the size of the team, you will implement one or more CAs of varying complexity, and design initial configurations that produce "interesting" results. These implementations will then be run with varying initial conditions, and generate time series displays of the CAs.

You are free to use C, C++, Python, or even your Arduino. Python in particular has proven in the past as perhaps the most efficient way to get decent working code (the overall goal). If you use Python, feel free to use the *time*, *csv*, *sys*, *copy*, *itertools*, and *pygame* modules. Use of any other modules requires preapproval of the instructor.

If you use your Arduino, the instructor has a few display units that can be loaned out, along with documentation on drivers.

2 History

A few events in the history of cellular automata:

- 1940s: Stanislaw Ulaw modeled crystal growth using a lattice network.
- 1940s: John von Neumann started studying self-replicating "robots" and generated a general paper on the subject[1] [10]
- 1950s: Ulam and von Neumann used 2D CA to calculate liquid motion.
- von Neumann developed a "universal copied and constructor" that had 29 state values per cell ([11] p. 848).
- von Neumann then demonstrated a 200,000 cell *tessellation model* that is now called a *von Neumann universal constructor*[2][3] [7].
- 1940s: Norbert Wiener and Arturo Rosenblueth used a cellular automaton-like model in studying cardiac systems that was converted to a true CA model by J. M. Greenberg and S. P. Hastings in 1978.
- 1960s: Cellular automata were used to develop mathematics of dynamical systems[4].
- 1969: Conrad Zuse proposes the universe is a CA ([8] p. 182)
- 1969: Alvy Ray Smith write a seminal PhD thesis that developed CA as a general class of computers[4] and led to an extensive survey [9][5].
- 1970: Martin Gardner popularizes The Game of Life [3]
- 1981-1990s: Stephen Wolfram used CA to help explain how complex patterns appear in nature in violation of 2nd Law of Thermodynamics, and then argued that CA has significance to all of science [11]
- 1984: The concept of *reversible CA* where the state update could be run backwards were shown useful in physics in general [6], and gas and fluid dynamics in particular ([5] p 379)

[1]See https://pdfs.semanticscholar.org/e853/8f11920fa6e56b3d34771bb330bd3e07281d.pdf
[2]paper available at http://cba.mit.edu/events/03.11.ASE/docs/VonNeumann.pdf
[3]see also https://en.wikipedia.org/wiki/Von_Neumann_universal_constructor
[4]see http://alvyray.com/Papers/PapersCA.htm for a list of papers
[5]Available at http://alvyray.com/Papers/CA/PolyCA76.pdf

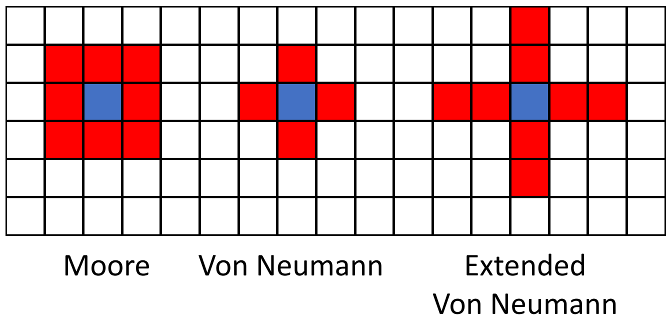

Figure 1: Common Cellular Automata Neighborhoods.

- 1987: Margolus and Toffoli start to develop architectures for hardware implementation of CAs useful for physics simulations [**?**]
- The concept of *totalistic CA* was introduced, where the state of a cell at some time step is an integer and depends only on the sum of the values of its surroundings at the prior time step. The Game of Life falls in this class.
- Probabilistic CA have rules that are probabilistic.
- Continuous Autoimata have state valuse that may be real numbers.
- 1998: Matthew Cook showed that a particular 1D CA (called "Rule 110") could be used as the basis of a universal TM [2].
- 2002: The Game of Life shown to be capable of emulating a Turing Machine [1][6]

3 Defining a Cellular Automata

A cellular automata is implemented on a possibly infinite grid of *cells*. While often square, each cell may take on other shapes such as hexagons, or even irregular as in "Penrose tiles."[7] The Game of Life assumes square cells.

3.1 State

At each time step each cell has a *state*: some value taken from some domain. This state value is updated simultaneously with that for all other cells at each time step. The Game of Life allows only two values for each cell's state: 0 or 1.

3.2 Rules

A set of rules defines how this state update is done, typically based on the state values of a "neighborhood" of each cell. Fig. 1 defines three common classes of neighborhoods for square cells in a 2D array. The Game of Life uses a Moore neighborhood.

In general, the rules for a state transition can be expressed as a table lookup where the index is formed from the values of the neighborhood and the current state, and the indexed value is the new state. For example, if the state values come from the set $\{0, 1\}$ and we are working with Moore neighborhoods, then there are $2^9 = 512$ entries in the table, with indexes from 000000000 to 111111111.

[6]http://www.igblan.free-online.co.uk/igblan/ca/

[7]Google "ppnrose tiles" for a wide variety of interesting patterns.

The Game of Life is an example of a *totalistic CA* where what goes into the table index is not the concatenation of neighboring state values but just their sum. For the Game of Life with state values of $\{0, 1\}$ this reduces the number of table entries from 512 to 18 (9 possible values for the neighborhood sum and 2 possible values for the current state value).

Besides the rules, most CAs also have an *initial configuration* to define the initial state. For the Game of Life this is typically a "0" for all but a few handful of cells that receive a "1." It is common to assume the gird is initially all one value (say "0"), and a list of cell indexes specifies what cells have different values.

3.3 Topology

Most theoretical CAs assume an infinite grid. Practically this is impossible to simulate, so a common alternative topology for 2D CAs is to assume a rectangular grid with location $(0, 0)$ being the top left cell, but the right neighbors of cells on the right of the grid are the cells in column 0, and the cells of column 0 have their left neighbors on the rightmost column. Likewise cells on the top and bottom rows have each other for upper and lower neighbors. This arrangement thus corresponds to a "donut."

3.4 1D CAs

An interesting special case of a CA is a 1D line of cells where the neighborhood of a cell is its two neighbors. For binary states there are 8 possible table entries. If each of these entries can be an arbitrary binary number there are thus $2^8 = 256$ different transition tables and thus 256 different CAs. These machines are known by which of the 256 tables are their table, called their *Wolfram Code*. Several of these machines exhibit particularly interesting behavior[8]. One of these (code 110 and known as "LeftLife") changes a cell state from 0 to 1 when the right neighbor is 1, and from 1 to 0 when both neighbors are 1.

Your simulator should be named **OneD-*team***, where *team* is the name of your team (netid if one-person team).

4 CAs to be Implemented

Depending on the size of the team, some number of different CAs are to be implemented as follows:

- A single student team will implement a 1D CA.
- A two-student team will implement both the above 1DCA and a 2D totalistic CA.
- A three-student team will implement both of the above, plus a 2D general CA.

In each case you should provide as input to your code the following:

- The number of cells in the CA.
- The initial configuration for the CA.
- The rules for updating the state of each cell at each time step.

The output shall be the CA state values as it evolves over time.

For teams of two or three students, a common *team-name* shall be used in all submissions.

[8]see https://en.wikipedia.org/wiki/Cellular_automatongutowitz and https://en.wikipedia.org/wiki/Elementary_cellular_automaton

	Current State Values			New State	
b	Left Neighbor	Current Cell	Right Neighbor	Rule 110	Rule 30
7	1	1	1	0	0
6	1	1	0	1	0
5	1	0	1	1	0
4	1	0	0	0	1
3	0	1	1	1	1
2	0	1	0	1	1
1	0	0	1	1	1
0	0	0	0	0	0

Table 1: Sample CA Transition Tables.

4.1 1D CA

A 1D CA simulator, shall assume a finite number of cells arranged in a linear fashion where the left neighbor of the rightmost cell is the leftmost cell, and vice versa.The domain of values for a cell's state will be $\{0, 1\}$. The specification of the rules for a specific CA shall be an integer from 0 to 255 in "Wolfram numbering" where if the machine number is an 8-bit integer $b = b_7b_6b_5b_4b_3b_2b_1b_0$ where if n is a 3-bit integer $n_2n_1n_0$ from 0 to 7, then b_n is the new state value when the left neighbor of the cell has a value n_2, the current state value is n_1, and the right neighbor has value n_0. Table 1 gives as an example of two rules of known interest.

4.1.1 Codes

Your simulator shall be defined in three parts. The first, to be named **oneD-define-*team*** where *team* is the name of the team, shall define the CA to be simulated, taking the following as input:

- The 8-bit Wolfram rule number defining the CA to be simulated.
- The length of the CA - at least 64 cells.

There should be two versions of the second code which will initialize the cell contents. The first of these, to be named **oneD-init-*team*** where *team* is the name of the team, shall take as an argument a list of cells to take in initial value of 1, and all other cells to take 0. The second of these, to be named **oneD-rand-*team*** where *team* is the name of the team, shall take as an input a seed value and generate a random assignment of 1's and 0's to the initial cell values. Giving this routine the same seed should always initialize the cells the same way.

The third code, to be named **oneD-run-*team***, shall accept a number of steps to be simulated, and when run, shall display the CA's complete state after that number of states. Ideally you should display the output for each of the intermediate time steps. It should be possible to call this routine multiple times, each time using initially the state of the CA from the last call.

4.1.2 Test Cases

You should experiment with different Wolfram rules, especially Rule 30 and Rule 110. You should try random initial configurations, and record those that seem to generate interesting behavior. For Rules 30 and 110 you might refer to

- https://en.wikipedia.org/wiki/File:CA_rule30s.png
- and https://upload.wikimedia.org/wikipedia/commons/thumb/f/fa/CA_rule110s.png/330px-CA_rule110s.png

For Rule 30 you should also try a run with a 1 on the rightmost cell, and see what happens.

From what you learn from looking at Rules 30 and 110, pick some other rules that may be interesting and describe what you see.

Each student should take responsibility for trying different machines.

4.2 TwoD Totalistic CA

The second CA shall be a 2D CA, again where the domain of state values is $\{0, 1\}$, and where the rules are expressed as a table indexed by the sum of the Moore neighbors (a number between 0 and 8). The 2D grid shall be wrapped on both sides as a donut.

Again there should be four codes:

- **twoDT-define-*team***: similar to **oneD-define-*team*** except that there are two arguments defining the size of the 2D grid, and the Rule argument shall be a 9-bit vector where bit i is the new state where the sum of the neighbors is i.
- **twoDT-init-*team***: similar to **oneD-init-*team*** except that the argument was a set of pairs of cell indices specifying an initial "1".
- **twoDT-rand-*team***: similar to **oneD-rand-*team***
- **twoDT-run-*team***: similar to **oneD-run-*team***

Running this CA should include developing a rule set that matches the Game of Life, and using configurations that correspond to known interesting patterns (see the sample patterns at https://bitstorm.org/gameoflife/ and http://www.math.cornell.edu/ lipa/mec/lesson6.html).

In addition, you should vary the rule set slightly are see which of the interesting configurations still hold.

Each student should take responsibility for trying one or more different machines.

4.3 TwoD General CA

The second CA shall be a 2D CA as in Section 4.2 but where the rules are expressed as a table indexed by all values of the Moore neighborhood. The 2D grid shall be wrapped on both sides as a donut.

Again there should be four codes:

- **twoD-define-*team***: similar to **twoDT-define-*team*** except that there are two arguments defining the size of the 2D grid, and the Rule argument shall be a table indexed by a 9-bit value consisting of the 8 neighbors and the current state.
- **twoD-init-*team***: similar to **twoDT-init-*team***.
- **twoD-rand-*team***: similar to **twoDT-rand-*team***
- **twoD-run-*team***: similar to **twoDT-run-*team***

It is expected that many of these programs are variants of those for the prior machine.

Running this CA should include developing a rule set that matches the Game of Life, and using configurations that correspond to known interesting patterns (see the sample patterns at https://bitstorm.org/gameoflife/ and http://www.math.cornell.edu/ lipa/mec/lesson6.html).

You may want to develop a simplified syntax for expressing the 512 entry rule table, such as only expressing conditions that cause the state to be a "1," and/or using a "*" for a particular neighbor column when either a 1 or 0 is needed. Make sure you describe your rule syntax.

In addition, you should vary the rule set slightly are see which of the interesting configurations still hold.

Each student should take responsibility for trying one or more different machines.

5 Documentation

Several types of documentation, all in PDF format, are required. One (entitled **readme-*team*.pdf** is submitted once by the entire team; a separate **teamwork-*netid*.pdf** is submitted separately by each member of the team. Finally, separate reports entitled **machine-name-*netid*.pdf** are to be submitted for each individual student-designed machine.

5.1 readme-team

A key part of what your teams submit is a **readme-*team*.pdf** that includes the following in this order:

1. The members of the team.
2. Approximately how much time was spent in total on the project, and how much by each student.
3. A description of how you managed the code development and testing. Use of github in particular is a particularly strong suggestion to simplifying this process, as is some sort of organized code review.
4. The language you used, and a list of libraries you invoked.
5. A description of the key data structures you used, especially for the internal representation of tapes, states, and the state machines and the transitions.
6. If you did any extra programs, or attempted any extra test cases, describe them separately.

Only one member of the team need submit this report to their Sakai project directory, along with the TM code.

5.2 Individual Machines

Each student shall submit in their own Sakai directory a copy of any machine that they personally designed and ran, the test files they created, and trace files of their execution. Also includedshall be a brief write-up, in a .pdf file of:

1. What problem the machine tackled.
2. What, if any, was the reference from which the problem solved by the machine was drawn.
3. How does the machine work, in words.

4. A state diagram or transition table.
5. How did you verify correct operation.

5.3 teamwork-netid

In addition, each team member should prepare a brief discussion of their own personal view of the team dynamics, and stored in a PDF called **teamwork-*netid*.pdf**, where netid is your netid. The contents should include:

1. Who were the other team members.
2. Under whose netid is the **readme-*team*.pdf**, code, and other material saved.
3. How much time did you personally spend on the project, and what did you do?
4. What did you personally learn from the project, both about the topic, above programming and code development techniques, and about algorithms.
5. In your own words, how did the team dynamics work? What could be improved? (e.g. did you use github and if so did it help, did you meet frequently enough, etc.)
6. From your own perspective, what was the role of each team member, and did any member exceed expectations, or vice versa.

Each student's submission here should be in their own words and SHOULD NOT be a copy of any other team members submission, nor should they be shared with the other team members. These reports will be kept private by the graders and instructor, and will be used to ensure healthy team dynamics. The instructor retains the right to adjust the score of an individual team member from the base score (both up and down) on the basis of these reports. Also, a composite of all such reports from all projects will be used to create an overall "lessons learned" at the end of the project in what techniques seemed to work better, and where problems arose. These hopefully will be of use for the next project.

6 Submission

- Each team member should have in their own Sakai directory for this course a directory called ProjectCA, where all submissions should go.
- When the team is ready to submit, one (and only one) team member will place copies of all code and test machine output in the designated common directory.
  - If you are using a traditional programming language (including Python) your code should be runnable on any of the studentnn.cse.nd.edu machines so that if there is an issue the graders can run the code themselves.
  - If you wrote in a compiled language like C++, include all needed source files (excepting standard libraries), a make file, and a compiled executable. The source code is there to allow the graders to look at the code for comments and to resolve any discrepancies that may arise in looking at your results.
  - If you wrote in a language like Python make sure your code is compatible with one of the versions supported on the studentnn.cse.nd.edu machines, again to allow the graders to check something if there is an issue.
  - If you implement your machines on the Arduino Due, upload all sketches to the Sakai directory in a form where they could be downloaded to a TA's Arduino and run there.

- Also in the same directory as the code the team should place an output file for each of the test files that you ran. The format should be as described in Section **??**, and the name should be the same name as the test file but with a "results-" prefix on the name.

In addition, every team member should include in their own ProjectCA Sakai directory:

- A copy of their individual **readme-*team*** file, again in pdf.
- Machine and test files for any designs that the student did themselves.
- Matching output files from running the above designs on the above test files.
- A short readme on the machine as described above.

7 Grading

Grading of the project will be based on a 100 points, divided up as the following

- Points off for late submissions (10 points per day).
- 5 points for following naming and submission conventions.
- 10 points for "reasonably" commented source code.
- 40 points based on the percent of cases you got correct for running the provided test problem.
- 20 points for completeness and quality of the readme file.
- 20 points for the student-designed machines and their tests.
- 5 points for the teamwork report.

All but the last two items will be common to all members of a team. The last two are specific to each member.

References

[1] Paul Chapman. Life universal computer. Nov. 2002.

[2] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15, 2004.

[3] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". In *Scientific American (223)*, pages 120–123, 1970.

[4] Gustav A. Hedlund. Endomorphisms and automorphisms of the shift dynamical system. In *Math. Systems Theory. 3 (4)*, pages 320–375, 1969.

[5] Jarkko Kari. Theory of cellular automata: A survey. *Theor. Comput. Sci.*, 334(1-3):3–33, April 2005.

[6] N. Margolus. Physics-like models of computation. *Physica D Nonlinear Phenomena*, 10:81–95, January 1984.

[7] John Von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, Champaign, IL, USA, 1966.

[8] Joel L. Schiff. *Cellular Automata: A Discrete View of the World (Wiley Series in Discrete Mathematics & Optimization).*

[9] Alvy Smith. Introduction to and survey of cellular automata or polyautomata theory1. 01 2018.

[10] John von Neumann. The general and logical theory of automata. In *Cerebral Mechanisms in Behavior - The Hixon Symposium 1948*, pages 1–31, 1951.

[11] Stephen Wolfram. *A New Kind of Science*. Wolfram Media Inc., Champaign, Ilinois, US, United States, 2002.