Networking Conway's Game of Life: Towards a dependency model for life and death

Jacob M. Peck
State University of New York, College at Oswego

Spring 2011

Introduction

The Game of Life, a popular cellular automaton ruleset introduced by John H. Conway in 1970, is a decent simulation of many things including as its name hints, life [1, 2]. However, this simulation appropriates only the effect of neighbors upon an individual. While this is accurate for the world of, say, bacterium, human (and to a certain extent, animal) lives have a much more intertwined structure [3]. Individuals are connected to far more than just their direct neighbors due to mobility, and to differing extents [3]. This project attempted to bridge the gap between the stationary cells of Conway's Game of Life and the mobile, active individuals in the world.

Project Description

To approach this problem, I devised a way to add connections to cells, with a probability of any given cell being a hub (and therefore having a significantly larger number of connections). These modified cells required a modified ruleset, which I decided to keep based on Conway's Game of Life, rather than any of the other two dimensional cellular automata rulesets solely out of personal interest and curiousity as to what would happen when a bit of network science was thrown into the mix.

In the end, I decided that each cell should rely on both its eight nearest neighbors and its world reaching connections for survival. In an attempt to simplify the simulation, and also in part due to time constraints, I decided that each cell would maintain its connections throughout the life of the simulation, meaning that once a cell, at say position `(0,3)` had a connection to the cell at position `(1,16)`, that connection persisted throughout the entire simulation, regardless of whether the cells are alive or dead. Also of note, connections are bi-directional, meaning that each cell involved in a connection is aware of the state of the other cell. In another act of simplification, hubs were decided to have up to ten connections, whereas non-hubs have at most two.

The modified ruleset I devised is as follows:

- A dead cell comes alive in the next generation if exactly 3 of its neighbors are alive in the current generation.
- A live cell dies in the next generation if it has more than 3 live neighbors in the current generation, as if by overpopulation.
- A live cell in the current generation with fewer than two live neighbors lives in the next generation if at least 30% of its neighbors are alive in the current generation, otherwise it dies, as if by isolation.

- All other cells remain dead in the next generation.

This allows for some interesting behavior, depending on the initial configuration parameters.

To facilitate the simulation of this process, I decided to write a program using the Processing programming language. Processing allows for easy programmatic control of a graphics surface, allowing for a visualization of what's going on behind the scenes.

I broke the project down into four steps, each building upon the steps that preceeded it.

1. Model Conway's Game of Life in it's standard form, allowing for a random initial configuration.
2. Add per-cell statistics, such as current lifespan, longest lifespan, average lifespan, lifespan history, and state history.
3. Add connections between cells, allowing for random distribution of hubs, while ensuring all connections are bi-directional and there are no self-referential connections.
4. Add per-cell statistics on hub status as well as a list of connections. Add system-wide statitstics on longest lives for hubs and non-hubs, as well as average lives for hubs and non-hubs.

This separation of tasks allowed me to quickly and efficiently produce a simulation that conformed to my requirements.

In modelling Conway's Life, I produced a simple simulator that ran Conway's life in a torroidal world (top cells stitched to bottom, left cells stitched to right, no cell has fewer than 8 neighbors). As a side effect of Processing's web-friendliness and its sister project Processing.js, I was able to port this simulation to the web [4]. This version will run in any HTML 5 compliant web browser.

The statistics are implemented in a way such that the system will print out a large text file after a predetermined number of generations have been run. Each cell in the text file will have an entry such as this:

```
Cell at (19,16):
   current lifespan: 0
   longest lifespan: 22
   average lifespan: 4.764045
   lifespan history: [17, 21, 8, 12, 5, 1, 2, 3, 5, 2, 6, 14, 1, 10, 10, 4, 1, 8, 2, 1,
8, 2, 4, 1, 5, 4, 5, 2, 1, 1, 10, 5, 1, 1, 3, 1, 1, 17, 1, 19, 22, 1, 6, 1, 8, 4, 2, 7,
1, 6, 3, 12, 6, 9, 3, 1, 9, 1, 3, 4, 1, 9, 1, 7, 2, 1, 2, 1, 6, 7, 1, 8, 6, 2, 2, 3, 3,
3, 4, 2, 7, 1, 4, 1, 1, 1, 2, 1, 1]
   memory: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 <...snip...>
```

The current lifespan refers to how long the cell had lived prior to statistics being collected, the longest lifespan is the longest amount of consecutive generations the sell was alive for, the average lifespan is the mean length of all of its lifespans, the lifespan history is a list of all the lifespans the cell went through, and the memory is a list containing either a `1` (denoting alive) or a `0` (denoting dead) for every single generation the system has gone through, sorted in chronological order. The above example has been truncated, as this was taken from a statistics file from a 1,000 generation run.

The connections are simply pointers to other cells. To visualize this, a line is drawn between any connected cells every time the system iterates. These lines are roughly color coded, so as to be unique and distinct. However, the end results are still terribly messy.

The system-wide statistics are just an addition to the text file produced by the previous statistics. Each cell is augmented by two additional fields, as follows:

```
Cell at (19,16):
   current lifespan: 0
   longest lifespan: 22
   average lifespan: 4.764045
   lifespan history: [17, 21, 8, 12, 5, 1, 2, 3, 5, 2, 6, 14, 1, 10, 10, 4, 1, 8, 2, 1,
8, 2, 4, 1, 5, 4, 5, 2, 1, 1, 10, 5, 1, 1, 3, 1, 1, 17, 1, 19, 22, 1, 6, 1, 8, 4, 2, 7,
1, 6, 3, 12, 6, 9, 3, 1, 9, 1, 3, 4, 1, 9, 1, 7, 2, 1, 2, 1, 6, 7, 1, 8, 6, 2, 2, 3, 3,
3, 4, 2, 7, 1, 4, 1, 1, 1, 2, 1, 1]
   memory: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 <...snip...>
   is hub: false
   connections: (8,11) (9,19)
```

The is hub statistic simply reports whether the given cell is a hub or not, and the connections list is simply the coordinates of the cells the given cell contains connections to.

The end of the statistics text file contains entries such as the following:

```
Average hub lifespan:     4.4231944
Average non-hub lifespan: 4.850761
Maximum hub lifespan:     72
Maximum non-hub lifespan: 80
```

This simply reports on the overall effectiveness of being a hub versus a non-hub.

Results

From several different runs, given different initial parameters, the statistics reported varied. Following is a table of results, with local maxima italicized.

world size	10 x 10	10 x 10	20 x 20	30 x 30
# of generations	100	100	1,000	10,000
hub density	2%	50%	20%	2%
max lifespan: hub	49	19	72	3,166
max lifespan: non-hub	*70*	*30*	*80*	*3,271*
average lifespan: hub	*12.2*	3.614888	4.4231944	*10.033964*
average lifespan: non-hub	7.788928	*4.0406003*	*4.850761*	9.810683
advantage	neither	non-hub	non-hub	neither

These results, while interesting, show no statistically relevant results at larger numbers of generations and larger worlds, however in smaller settings within a constrained time period, a minor advantage is shown in favor of the non-hub cells. In both cases where there is no outright advantage (both a longer average lifespan and a longer maximum lifespan), there is a common pattern whereby a single non-hub cell had an outstanding life, but a hub cell can expect to live longer on average.

What I Learned

In interepreting these results, I have come to a few realizations. Firstly, that a non-hub cell is favored in several ways for these runs, it stands to reason that it could simply be that knowing more people complicates life. By requiring 30% live connections to prevent a cell from dying by isolation, non-hubs have an advantage, as they only need one connection alive to continue into the next generation, whereas a hub needs at least three. While the burden of responsibility is lightened a bit by having more choices, it is still a stricter requirement. In a way, this maps to actual life, as well. Though there exist hubs, and they have a large number of connections, they have more people to keep happy and may very well operate at a higher stress level than non-hub individuals.

It also speaks of the unity of the human species. If a well connected, socially adept hub individual has just as much chance to make it as a non-hub individual, it shows that there's not truly much difference between us. The people we know and the choices we make are fundamentally similar, across borders, genders, and races.

Outside of the network science aspects I was able to draw out of this, I also became acquainted with the Processing programming language a bit more than I had been beforehand, and also improved my implementation ideas of Conway's Life. Having implemented Conway's life three times in the year previous to this project, I had a firm foundation to start with, but this project simplified my implementation of the torroidal aspect of the world, as well as with speed.

In future versions of this project, I would like to add more aspects to study, such as enemy connections that work against a cell, aging in the form of different cell states, economics (wealth), social status and mobility (ability to form new friend connections and prevent enemy connections), and possibly inheritance (cell offspring gaining the wealth and connections from their parents). This project seems to have a very open future, and I hope I am able to see it through.

References

[1] *Conway's Game of Life*, LifeWiki, `http://www.conwaylife.com/wiki/index.php?title=Conway%27s_Game_of_Life`, Accessed 09 May 2011.

[2] Stephen Wolfram, *A New Kind of Science*, Wolfram Media, Inc., 2002.

[3] Duncan J. Watts, *Six Degrees: The Science of a Connected Age*, W. W. Norton & Company, Inc., 2003.

[4] Jacob M. Peck, *Conway's Game of Life {2011}*, suspended-chord:portfolio, `http://suspended-chord.info/portfolio/programming/conways-game-of-life-2011/`, 2011.

Appendix - Code Listings

What follows is a listing of the code for this project as it stood at the time of writing.

Listing of NetworkedGoL.pde

```
// Hon 301 Final Project - Networked Game of Life
// Main
// Hon 301 - Vampola
// Jacob Peck

color live_color = 255; // white
color dead_color = 0; // black
color stroke_color = color(155); // gray, this is the color between cells


int cellsize = 20; // height and width (in pixels) of each cell
int x = 20; // system width in cells
int y = 20; // system height in cells
float density = 0.5; // percentage of live cells in the initial configuration
float hubdensity = 0.02; // percentage of hubs

int generationcount = 1000; // generations to run before writing statistics (stats.txt)

CA ca;
int count = 0;

void setup(){
  // do setup stuff here
  println("starting...");
  size(x*cellsize, y*cellsize);
  background(dead_color);
  stroke(stroke_color);
  //noStroke(); // uncomment this to remove the grid
  ca = new CA(x,y,density,hubdensity,cellsize,live_color,dead_color);
  ca.drawCA();
}



void draw(){
  // do looping stuff here
  //delay(90); // uncomment this to add a slight delay between iterations
  if (count < generationcount) {
    stroke(stroke_color);
    ca.iterate();
    ca.drawCA();
    count++;
```

```
  } else {
    ca.printStats(); // comment this to disable printing stats
    exit();
  }
}
```

Listing of Cell.pde

```
// Hon 301 Final Project - Networked Game of Life
// Cell class
// Hon 301 - Vampola
// Jacob Peck

class Cell {
  private int x;
  private int y;

  private boolean state;
  private ArrayList<Cell> neighbors;

  private ArrayList<Cell> connections;
  private boolean hub;

  private int currentLifespan = 0;
  private ArrayList<Integer> lifespanHistory;
  private ArrayList<Boolean> memory;

  // constructor
  public Cell(int x, int y, boolean state, boolean hub) {
    this.x = x;
    this.y = y;
    this.state = state;
    this.hub = hub;

    connections = new ArrayList<Cell>();
    neighbors = new ArrayList<Cell>();

    lifespanHistory = new ArrayList<Integer>();
    memory = new ArrayList<Boolean>();
    memory.add(state);

    if (state) currentLifespan = 1;
  }


  // getters
  public int getX() {
    return x;
  }

```

```java
  public int getY() {
    return y;
  }

  public boolean getState() {
    return state;
  }

  public ArrayList<Cell> getConnections() {
    return connections;
  }

  public ArrayList<Cell> getNeighbors() {
    return neighbors;
  }

  public int getCurrentLifespan() {
    return currentLifespan;
  }

  public ArrayList<Integer> getLifespanHistory() {
    return lifespanHistory;
  }

  public ArrayList<Boolean> getMemory() {
    return memory;
  }

  public boolean isHub() {
    return hub;
  }

  // setters
  public void setState(boolean newState) {
    state = newState;
  }

  public void setCurrentLifespan(int value) {
    currentLifespan = value;
  }

  public void setHub(boolean hub) {
    this.hub = hub;
  }

  // various functionality
  public void addNeighbor(Cell neighbor) {
    if (neighbors.size() >= 8 || neighbors.contains(neighbor))
      return;
```

```
    else
      neighbors.add(neighbor);
  }

  public void addConnection(Cell connection) {
    if (connections.contains(connection) || (connection.getX() == x &&
        connection.getY() == y))
      return;
    else
      connections.add(connection);
  }

  // ** not utilized **
  public void removeConnection(Cell connection) {
    if (connections.contains(connection))
      connections.remove(connection);
  }

  public int getLiveNeighborCount() {
    int count = 0;
    for (Cell c : neighbors)
      if (c.getPreviousState())
        count++;
    return count;
  }

  public int getLiveConnectionCount() {
    int count = 0;
    for (Cell c : connections)
      if (c.getPreviousState())
        count++;
    return count;
  }

  public boolean getPreviousState() {
    return memory.get(memory.size()-1);
  }

  private void iterate() {
    int count = getLiveNeighborCount();
    float connectionpercentage =
        float(getLiveConnectionCount()/getConnections().size());
    if (getState()) { // live, survives at 2 or 3 neighbors, or failing due to
        underpopulation that, at least 30% live connections
      if (count == 2 || count == 3 || (count < 2 && connectionpercentage >= 0.3)) {
        incrementCurrentLifespan();
      } else {
        setState(false);
        //setState(true);
```

```
        // switch the commenting on the above two lines to enable life without death (B3/S12345678)
        startNewLife();
      }
    } else { // dead, born at 3 live neighbors
      if (count == 3) {
        setState(true);
        incrementCurrentLifespan();
      }
    }
    addToMemory();
  }

  // add the current state to the memory
  public void addToMemory() {
    memory.add(state);
  }

  // increment life counter by one
  public void incrementCurrentLifespan() {
    currentLifespan++;
  }

  //
  public void startNewLife() {
    if (currentLifespan > 0) lifespanHistory.add(currentLifespan);
    currentLifespan = 0;
  }
}
```

Listing of CA.pde

```
// Hon 301 Final Project - Networked Game of Life
// CA class
// Hon 301 - Vampola
// Jacob Peck

class CA {
  private Cell[][] cells;
  private int x;
  private int y;
  private int cellsize;
  private color color_live;
  private color color_dead;

  // constructor, sets up the parameters
  public CA(int x, int y, float density, float hubdensity, int cellsize, color color_live, color color_dead) {
    println("  making ca...");
    this.x = x;
```

```
    this.y = y;
    this.cellsize = cellsize;
    this.color_live = color_live;
    this.color_dead = color_dead;

    cells = new Cell[x][y];

    initializeCells(density, hubdensity);
  }

  // initializes cells, randomly distributes live cells according to density
  private void initializeCells(float density, float hubdensity) {
    print("    initializing cells   ");
    for (int i = 0; i < x; i++) {
      for (int j = 0; j < y; j++) {
        print(".");
        boolean hub = (random(1) <= hubdensity);
        boolean live = (random(1) <= density);
        cells[i][j] = new Cell(i, j, live, hub);
      }
    }
    println();
    assignNeighbors();
    assignConnections();
  }

  // assigns all neighbors to a cell
  private void assignNeighbors() {
    print("    assigning neighbors  ");
    for (int i = 0; i < x; i++) {
      for (int j = 0; j < y; j++) {
        print(".");
        assignNeighborToSingleCell(i, j, i-1, j-1);
        assignNeighborToSingleCell(i, j, i, j-1);
        assignNeighborToSingleCell(i, j, i+1, j-1);

        assignNeighborToSingleCell(i, j, i-1, j);
        assignNeighborToSingleCell(i, j, i+1, j);

        assignNeighborToSingleCell(i, j, i-1, j+1);
        assignNeighborToSingleCell(i, j, i, j+1);
        assignNeighborToSingleCell(i, j, i+1, j+1);

      }
    }
    println();
  }

  // assigns a number of unique bi-directional connections between cells
```

```
private void assignConnections() {
  print("    assigning connections");
  int nonhubconnections = 2;
  int hubconnections = 10;

  int numconnections;
  for (int i = 0; i < x; i++) {
    for (int j = 0; j < y; j++) {
      print(".");
      if (cells[i][j].isHub()) numconnections = hubconnections;
      else numconnections = nonhubconnections;

      int loopcount = 0;

      while (cells[i][j].getConnections().size() < numconnections && loopcount < 1000) { // prevent infinite loops if impossible to find another unique connection
        loopcount++;

        // pick random cell
        int connx = int(random(x));
        int conny = int(random(y));

        // ensure that it can accept the connection as well
        int targetconncount = cells[connx][conny].getConnections().size();
        if ((cells[connx][conny].isHub() && targetconncount < hubconnections) || targetconncount < nonhubconnections) {
          cells[i][j].addConnection(cells[connx][conny]);
          cells[connx][conny].addConnection(cells[i][j]);
        }
      }
    }
  }
  println();
}

// ensures that a neighbor is assigned torroidally
private void assignNeighborToSingleCell(int x, int y, int x2, int y2) {
  if (x2 > this.x - 1) x2 = 0;
  if (x2 < 0) x2 = this.x - 1;
  if (y2 > this.y - 1) y2 = 0;
  if (y2 < 0) y2 = this.y - 1;
  cells[x][y].addNeighbor(cells[x2][y2]);
}

// iterates the entire system, one cell at a time.
public void iterate() {
  for (int i = 0; i < x; i++) {
    for (int j = 0; j < y; j++) {
```

```
        cells[i][j].iterate();
      }
    }
  }

  // draws the CA to the screen
  private void drawCA() {
    // wipe screen
    background(color_dead);

    for (int i = 0; i < x; i++) {
      for (int j = 0; j < y; j++) {
        // find position
        int posx = i * cellsize;
        int posy = j * cellsize;

        // select state color
        if (cells[i][j].getState())
          fill(color_live);
        else
          fill(color_dead);

        // draw cell
        rect(posx, posy, cellsize, cellsize);
      }
    }

    drawConnections();
  }

  // draws the connections between the cells to the screen (ugly)
  private void drawConnections() {
    for (int i = 0; i < x; i++) {
      for (int j = 0; j < y; j++) {
        int hashcode = cells[i][j].hashCode();
        color temp = color(red(hashcode),green(hashcode),blue(hashcode)); //
            pseudo-unique colors (based on source cell)
        fill(temp);
        stroke(temp); // consistent colors per connection

        int startx = i * cellsize + (cellsize/2);
        int starty = j * cellsize + (cellsize/2);
        ellipse(startx, starty, (cellsize/4), (cellsize/4));

        for (Cell c : cells[i][j].getConnections()) {
          if (c.getX() < i && c.getY() < j) continue; // prevent double drawing
          int endx = c.getX() * cellsize + (cellsize/2);
          int endy = c.getY() * cellsize + (cellsize/2);

```

```
          line(startx, starty, endx, endy);
        }
      }
    }
  }

  private void printStats() {
    PrintWriter output = createWriter("stats.txt");
    output.println("Stats for run (" + year() + "-" + month() + "-"  + day() + " " + hour() + ":" + minute() + ":" + second() + ")");
    int maxnonhublifespan = 0;
    int maxhublifespan = 0;

    float avg_hublifespan = 0;
    float avg_nonhublifespan = 0;

    float hubcount = 0;
    float nonhubcount = 0;

    for (int i = 0; i < x; i++) {
      for (int j = 0; j < y; j++) {
        int current = cells[i][j].getCurrentLifespan();

        cells[i][j].startNewLife();

        int max_lifespan = 0;
        float avg_lifespan = 0;
        String memory = "";
        String connections = "";
        boolean is_hub = cells[i][j].isHub();

        // longest lifespan + average lifespan
        for (Integer k : cells[i][j].getLifespanHistory()) {
          avg_lifespan += k;
          if (k > max_lifespan) max_lifespan = k;
        }
        avg_lifespan /= (float)cells[i][j].getLifespanHistory().size();
        if (cells[i][j].isHub()) {
          if (max_lifespan > maxhublifespan) maxhublifespan = max_lifespan;
          hubcount++;
          avg_hublifespan += avg_lifespan;
        } else {
          if (max_lifespan > maxnonhublifespan) maxnonhublifespan = max_lifespan;
          nonhubcount++;
          avg_nonhublifespan += avg_lifespan;
        }

        // memory
        for (Boolean b : cells[i][j].getMemory()) {
```

```java
          if (b) memory = memory + "1 ";
          else memory = memory + "0 ";
        }
        memory = memory.trim();

        // connections
        for (Cell c : cells[i][j].getConnections())
          connections = connections + "(" + c.getX() + "," + c.getY() + ") ";
        connections = connections.trim();

        output.println("Cell at (" + i + "," + j + "):");
        output.println("   current lifespan: " + current);
        output.println("   longest lifespan: " + max_lifespan);
        output.println("   average lifespan: " + avg_lifespan);
        output.println("   lifespan history: " + cells[i][j].getLifespanHistory());
        output.println("   memory: " + memory);
        output.println("   is hub: " + is_hub);
        output.println("   connections: " + connections);
        output.println("\n\n");

      }
    }
    output.println("Average hub lifespan:     " + (avg_hublifespan / hubcount));
    output.println("Average non-hub lifespan: " + (avg_nonhublifespan / nonhubcount));
    output.println("Maximum hub lifespan:     " + maxhublifespan);
    output.println("Maximum non-hub lifespan: " + maxnonhublifespan);

    output.flush();
    output.close();
  }
}
```