Growing Simulated Bacteria Colonies with Cellular Automata

Thomas Guerena 11/22/2015

Abstract

This project is a naive simulation of bacteria colony growth and evolution. The environment is confined to a Z_n x Z_n plane, the passage of time has been warped to both slow the rate of bacteria growth and to accelerate the rate of genetic diversity, and exposes thirteen parameters for users to adjust. The purpose of this project is to provide people with non-computer science backgrounds a tool for exploring cellular automata. The design does not aim for realism, but instead to maximize the visual effects of a user's input.

All code, proposals, and project reports can be found on github.
github.com/thomasguerena/cs441-ai-project

Project Evolution

That was the only pun in this report.

This project has undergone a number of significant revisions. The final product is not a fair representation of all the work completed, nor the knowledge I've gained. Here is an outline of the various versions of my project, and what I took away from each.

Initial Project Design

In the beginning, I wanted to use co-evolutionary programming and genetic algorithms to show how bacteria could evolve to resist an antibiotic, using cellular automata. The purpose was to show the difference in rates of evolution between bacteria which use horizontal gene transfer, and those that don't. Despite running into performance issues when expanding my environment (I'll come back to this), I successfully implemented this in the early weeks of the project. Despite creating an algorithm that could very successfully defeat even the toughest antibiotics I created, it wasn't interesting to watch. My visualization of the process showed chaotic bacteria interactions, and the result was the same every time - convergence to the ideal genetic coding. At first, I thought I had created something very intelligent. One of my favorite explanations of intelligence is that the more intelligent a system the harder its methods are to predict, but the more predictable its outcome. For example, you cannot predict the moves of a master chess

player, but you can say with certainty you will not win. However, my simulation turned out only to be a glorified hill-climbing problem. My implementation wasn't incorrect, it was just a boring problem to solve. I wanted emergent behavior similar to Conway's Game of Life, but pertaining to my problem.

Second Major Project Design

Disappointed but undeterred by the results of my first few weeks of development, I decided the solution to my predictable bacteria was to give each one independence. Rather than each bacteria acting under the same rules, I created a small genetic encoding which defined an individual's behavior given any one of more than 13,000 neighborhood states (3x3). As bacteria mated and mutated, changes were made to their genetic code, thereby changing their behavior. It worked, but there was so much genetic and behavioral diversity that the simulation again looked chaotic. It was impossible to distinguish the actions and decision making of any of the bacteria. Again I ran into performance issues and mysterious steady-states that even generations of genetic mutation could not solve. An even more fundamental issue was that I had no method for evaluating fitness in my bacteria colony at run time. I could evaluate the fitness of a single bacteria by putting it in each possible neighborhood state and tracking how often it survived, and how often it died. This would give me a general fitness for that genetic code, but unless I wanted to create a table of each possible genetic code and its fitness, it wasn't a viable runtime solution. Instead, I abandoned the concept and began another redesign.

Final Project Design

Frustrated that I was so far unable to create anything visually intelligible, I gave up on the idea of creating an emergent CA and decided to focus on a CA simulation that was as easy to watch and understand as possible. I thought I could achieve this by focusing on the following rules:

1. Behavior rules apply globally. Entity types do not necessarily share behaviors, but entities of the same type (e.g. bacteria) must behave identically in identical states.
2. Use as simple of behavior rules as possible.
3. Use little to no randomization, for any purpose. Deterministic simulations are easier to watch and understand.
4. Runtime fitness is a must.
5. Keep populations low. By slowing down the rate of replication, you free up a lot of physical space in the environment. More space means an individual's actions are not often limited by the number of adjacent empty matrix cells. This was a real problem in my previous two designs.
6. **Create long term goals**. Up until this point, each bacteria had made decisions about their actions every generation. If their decisions change each generation (as is almost certain given their fluctuating environment) then their behavior has no continuity. Long term goals allow observers to track an individual's actions over time and infer the goal or motivation for the behavior. This became the most important aspect of my design.

These rules resulted in a simulation that was *much*, *much* easier to read and more fun to interact with. I will explain my design changes in more detail, but suffice it to say that observers could easily see which action bacteria are taking, when they decide to change actions, and why those actions changed. It's typically clear (especially to me, now that I've run the simulation thousands of times) what the current goal of any one bacteria is at any time, and how they will likely act in the next generation. However, the simulation is not predictable. Like Conway's Game of Life, given any start state, it's impossible to determine with certainty how to the simulation will end (or if it will end at all).

Details (Final Design)

- **Parameterized simulation**. Almost every aspect of the simulation is built to be adjustable. Currently, only 13 of these parameters are exposed to the user. This was just to keep the interactions between the user and simulation simple. I wanted each user adjustment to feel impactful, and I think providing too many knobs to adjust would leave people feeling overwhelmed and confused about how they were changing the simulation.
- **Energy**. Each bacteria now has an associated energy level. An individual can spend this energy to take actions, such as move, replicate, or mate. When an individual falls to zero energy, they die.
- **Energy oriented goals**. Bacteria have set goals - eat, mate, replicate - which are ordered in a priority queue. Breakpoints are set to create thresholds between goals. For example, a user can order bacteria goals (from lowest priority to highest) as *mate*, *eat*, *replicate*. In this example, the energy thresholds might by 33 and 66. Meaning, the bacteria would try to mate if it had more than 66 energy, try to eat if it had between 66 and 33 energy, and try to replicate if it had less than 33 energy. These priorities and thresholds are set globally, so all bacteria behavior according to the same rules. During the simulation, however, bacteria do not have uniform energy levels, so many different goals and actions are shown on screen at once.
- **Food**. A new entity was added to allow bacteria to regain energy. Each food entity has some number of *sustenance* to offer. Bacteria currently trying to eat will search out food as best they can, and then stop and eat until either the food runs out, or their energy levels rise high enough that their priorities shift.
- **Diversity**. Any concept of DNA or advanced genetic behavioral encoding was replaced by a simple integer representation of genetic diversity. Further research on the biological processes in which bacteria come to resist antibiotics showed that this general occurs when bacteria achieve a genetic diversity great enough that the vulnerabilities targeted by the antibiotic have either been erased, or the function of the bacteria is such that those vulnerabilities no longer affect the functions of the bacteria. To model this, I simply increase a bacteria's *diversity* each time it mates or mutates. Mutation occurs randomly (at a rate set by the user), and provides a minor diversity increase. Mating is more expensive and requires searching for a partner, but provides a more significant increase to diversity.

Project Statistics

Portion of the proposed project completed...

This is hard to gauge as the project design changed pretty drastically. However, I would say that since I coded each version of my project to near or total completion, I will say that 50% of my initial design was completed (I completed it, but did not port it to Python as stated in my proposal), 75% of my second design (I abandoned it before dealing with many of its outstanding issues), and 100% of my final design.

Code Statistics

Github says I added 4,828 lines and removed 2,204 - so as of today, my net lines is about 2,500. However, I would conservatively guess that 1,000 of those are from libraries and third-party UI styles. All code save for an icon set and a utility library were written and tested by me. All functions and modules were testing independently before integrating, and then again as an integrated system. However, due to the frequent design changes, I have no formal unit or integration tests.

Unanticipated Obstacles

Most of my unanticipated obstacles were in the design process, not implementation. My multiple redesigns definitely ate up a lot of time, but they also allowed me to gain a broader understanding of the different approaches to cellular automata, and a deeper understanding of how design decisions and modeling affect the behavior of the automata. I feel that those obstacles were sufficiently discussed in the sections above.

Pertaining to performance, I ran into a number of language specific limitations. My prototype was written in Javascript. However, as the project evolved and shifted into a semi-eduction focused web application, that prototype became my main code base. This consequences of this included no support for vectorized implementations of matrices. In Javascript, arrays are implemented as lists, and arrays longer than 256 are converted to multiple arrays. For example, *new Array(400)* creates an “array” which is stored in a 256-list and a 144-list. Basically, I was stuck with a two-dimensional matrix implementation no matter how I coded it. This slowed down a lot of my algorithms and limited the maximum size of my environment. This was especially evident in my first project design. But ultimately, the performance didn’t matter once I changed my priorities to user readability.

Experience Gained

- Genetic Algorithms

- Evolutionary and Coevolutionary programming
- Cellular Automata

I gained the first-hand experience required to really understand why there's as much art to these algorithms as there is science. I found that building a general framework for solving these problems was easy - it was tweaking my representation that really took a long time to solidify. I learned that your model should be simple and deterministic, and that abstract representation is more advantageous than literal representations. Also, you can't encode the freedom required for emergent behaviors into your automata. Instead, you have to encode a very small, very strict set of rules, and then leave as much negative space as possible to allow those rules to emerge as interesting behaviors. This probably isn't true for all CA simulations, but certainly for this case. And I imagine the more complex the problem, the larger the emphasis on simplicity in representation.

Project Code

All code below was written myself. This code includes only the logic pertaining to the simulation. Any code for the UI, styles, markup, etc. has been left out. Again, the project is available in its entirety on github: github.com/thomasguerena/cs441-ai-project

Build

The easiest way to run the code is to clone the repository (or download a zip package of the repository) and simply open *index.html* in Chrome. There should be no need to build anything, as the built version of the application is currently on github. If for some reason you want to make changes to the application and need to rebuild, follow these instructions:

If changes were made to any javascript files, simply refresh *index.html* and those changes will appear.

If changes were made to any style/SASS files, follow these steps.

1. Install nodejs, which now comes with npm by default: nodejs.org
2. Install bower: *npm install -g bower*
3. Install grunt: *npm install -g grunt*
4. Install grunt-cli: *npm install -g grunt-cli*
5. Build: *grunt sass*
6. (optional) Watch files to automatically rebuild: *grunt watch*

Appendix

The order in which the code is presented is as follows:

1. Entity classes - Bacteria, Antibiotic, Food
2. Environment class
3. Main simulation
4. Canvas drawing
5. Settings
6. Utilities

Bacteria

```
(function () {
	'use strict';

	window.Bacteria = function (x, y, diversity) {

		var that = this;
		this.age = 0; // number of generations survived
		this.energy = settings.bacteria.energy;
		this.diversity = diversity > -1 ? diversity : 0;
		this.actions = { // goal-oriented functions stored by goal
			food: function () { that.eat(); }, // search for food, eat food
			mate: function () { that.mate(); }, // search for mate, mate
			replicate: function () { that.replicate(); } // replication
		};
		this.goal = 1; // default to first priority
		// FIXME - doesn't handle collisions
		this.x = x > -1 ? x : Math.floor(Math.random()*environment.w);
		this.y = y > -1 ? y : Math.floor(Math.random()*environment.h);
		this.heading = { x: 0, y: 0 };
		while (this.heading.x == 0 && this.heading.y == 0) {
			// Random headings are fine so long as food can be detected within
			// some non-adjacent proximity.
			this.heading = {
				x: Math.random() > 0.5 ? -Math.round(Math.random()) :
Math.round(Math.random()),
				y: Math.random() > 0.5 ? -Math.round(Math.random()) :
Math.round(Math.random())
			};
		}
	};

	Bacteria.prototype.yieldGoal = function () {
		++this.goal;
		if (this.goal == 2) {
			this.actions[settings.bacteria.priorities.second]();
		}
		else if (this.goal == 3) {
			this.actions[settings.bacteria.priorities.third]();
		}
```

```
};

Bacteria.prototype.resetGoal = function () {
	this.goal = 1;
};

Bacteria.prototype.update = function () {

	this.age += 1; // age one generation
	this.resetGoal(); // reset actions to default
	this.mutate(); // potentially mutate

	// Consider possible actions
	if (this.energy < settings.bacteria.thresholds.lower) {
		this.actions[settings.bacteria.priorities.first]();
	}
	else if (this.energy < settings.bacteria.thresholds.upper) {
		this.actions[settings.bacteria.priorities.second]();
	}
	else {
		this.actions[settings.bacteria.priorities.third]();
	}

	if (this.energy < 1) environment.remove(this);
};

Bacteria.prototype.move = function () {
	var that = this;
	var emptyAdj = null;
	var availableMoves = null;

	if (this.energy < settings.bacteria.cost.move) this.yieldGoal();

	emptyAdj = environment.getEmptyAdjacent(this);
	availableMoves = emptyAdj.bacteria.intersect(emptyAdj.food,
		function (a, b) {
			return a.x == b.x && a.y == b.y;
	});

	if (availableMoves.length > 0) {
		environment.remove(this);

		if (!availableMoves.find(function (avail) {
			return avail.x == that.x + that.heading.x
				&& avail.y == that.y + that.heading.y;
		})) {
			var r = Math.floor(Math.random()*availableMoves.length);
			this.x = availableMoves[r].x;
			this.y = availableMoves[r].y;
		} else {
```

```
                        this.x += this.heading.x;
                        this.y += this.heading.y;
                }
                this.energy -= settings.bacteria.cost.move;
                environment.add(this);
        } else {
                this.yieldGoal();
        }
}

Bacteria.prototype.eat = function () {

        var adjFood = environment.getAdjacent(this).food;
        var nearbyFood = null;

        if (adjFood.length == 0) {
                this.move();
                nearbyFood = this.senseFood();
                if (nearbyFood) {
                        this.heading.x = nearbyFood.x > this.x ? 1 : -1;
                        this.heading.y = nearbyFood.y > this.y ? 1 : -1;
                }
        } else {
                this.energy += adjFood[0].munch();
        }
};

Bacteria.prototype.mate = function (potentialMates) {

        var adjBacteria = null;

        if (this.energy < settings.bacteria.cost.mate) this.yieldGoal();

        adjBacteria = environment.getAdjacent(this).bacteria;

        if (adjBacteria.length > 0) {
                this.energy -= settings.bacteria.cost.mate;
                this.diversity += 10; // TODO - maybe this should depend on the other's
diversity
        }
        else {
                this.yieldGoal();
        }
};

Bacteria.prototype.replicate = function () {
        var rx = -1;
        var ry = -1;
        var rd = this.diversity;
        var emptyAdj = null;
```

```
                if (this.energy < settings.bacteria.cost.replicate) this.yieldGoal();

                emptyAdj = environment.getEmptyAdjacent(this).bacteria;

                if (emptyAdj.length > 0) {
                        var i = Math.floor(Math.random()*emptyAdj.length);
                        rx = emptyAdj[i].x;
                        ry = emptyAdj[i].y;
                        environment.add(new Bacteria(rx, ry, rd));
                        this.energy -= settings.bacteria.cost.replicate;
                } else {
                        this.yieldGoal();
                }
        };

        Bacteria.prototype.mutate = function () {
                if (Math.floor(Math.random()*100) < settings.bacteria.mutationRate) {
                        this.diversity += settings.bacteria.mutationStep;
                }
        };

        /**
         * @returns {Obj} xy-coordinate pair of a nearby food cell.
        */
        Bacteria.prototype.senseFood = function () {
                // By getting adjacent food from one cell north,
                // south, east, and west of our current location,
                // we effectively search radially outward. This
                // does search some cells mutiple times, but the
                // performance differences should be unnoticable.
                var north = environment.getAdjacent({ x: this.x, y: this.y + 1 }).food;
                var south = environment.getAdjacent({ x: this.x, y: this.y - 1 }).food;
                var east = environment.getAdjacent({ x: this.x + 1, y: this.y }).food;
                var west = environment.getAdjacent({ x: this.x - 1, y: this.y }).food;
                var nearbyFood = north.concat(south, east, west);

                if (nearbyFood.length) return nearbyFood[0];
                else return null;
        };
})();
```

Antibiotic

```
(function () {
        'use strict';

        window.Antibiotic = function (x, y, potency) {
         this.x = x > -1 ? x : Math.floor(Math.random()*16);
```

```
        this.y = y > -1 ? y : Math.floor(Math.random()*16);
        this.potency = potency > -1 ? potency : settings.antibiotic.potency;
        this.diffused = false;
       };

    Antibiotic.prototype.spread = function () {
        if (generation < settings.antibiotic.genx) return;
        if (this.diffused == true) return;
        if (this.potency - settings.antibiotic.diffusal < 0) return;

        var emptyAdj = environment.getEmptyAdjacent(this);
        var availableSpread = emptyAdj.antibiotic.intersect(emptyAdj.food,
            function (a, b) {
                return a.x == b.x && a.y == b.y;
        });

        for (var i = 0; i < availableSpread.length; ++i) {
            environment.add(new Antibiotic(
                availableSpread[i].x,
                availableSpread[i].y,
                this.potency - settings.antibiotic.diffusal
            ));
            this.diffused = true;
        }
    };

    /* @description Decides whether or not it kills
     *   a bacteria cell. If it doesn't kill the cell,
     *   this antibiotic will be destroyed.
     * @oaram {Bacteria} cell
     * @returns {Boolean}
     *   True: It kills the bacteria.
     *   False: It does NOT kill the bacteria.
    */
       Antibiotic.prototype.kills = function(cell) {
              return this.potency > cell.diversity;
       };
})();
```

Food

```
(function () {
       'use strict';

       window.Food = function (x, y) {
              this.x = x;
              this.y = y;
              this.sustenance = settings.food.sustenance;
```

```
        if (this.x == undefined || this.y == undefined) {
                var rx = new Array(environment.w);
                var ry = new Array(environment.h);
                for (var i = 0; i < environment.w; ++i) rx[i] = i;
                for (var i = 0; i < environment.h; ++i) ry[i] = i;
                rx = knuthShuffle(rx);
                ry = knuthShuffle(ry);

                for (var i = 0; i < rx.length; ++i) {
                        for (var j = 0; j < ry.length; ++j) {
                                if (environment.antibioticMatrix[rx[i]][ry[j]] < 0
                                 && environment.bacteriaMatrix[rx[i]][ry[j]] < 0
                                 && environment.foodMatrix[rx[i]][ry[j]] < 0) {
                                        this.x = rx[i];
                                        this.y = ry[j];
                                        return;
                                }
                        }
                }
        }
};

Food.prototype.munch = function() {
        var bitesize = 10; // not BYTE size...
        this.sustenance -= bitesize;
        return bitesize;
};

Food.prototype.update = function () {
        if (this.sustenance < 1) {
                environment.remove(this);
        }
};

})();
```

Environment

```
(function () {
        'use strict';

        /*-- Environment --*/

        /* @description Manages all instances of bacteria and antibiotic
         *   in the simulation environment, analogous to a petri dish.
        */
        window.Environment = function () {

                // Default dimensions
```

```
        this.w = settings.environment.width;
        this.h = settings.environment.height;

        // Fixed-length arrays containing bacteria
        //   antibiotic, and food objects, respectively.
        this.bacteriaList = new Array(this.w*this.h);
        this.antibioticList = new Array(this.w*this.h);
        this.foodList = new Array(this.w*this.h);

        // Integer matrices holding the index at which
        //   the bacteria, antibiotic, or food can be
        //   found in their respective lists.
        this.bacteriaMatrix = [];
        this.antibioticMatrix = [];
        this.foodMatrix = [];

        // Population trackers
        this.bacteriaCount = 0;
        this.antibioticCount = 0;
        this.foodCount = 0;

        // Behavior trackers
        this.antibioticDiffusion = 1; // arbitrary non-zero initial value

        // Construct matrices
        for (var i = 0; i < this.w; ++i) {
                this.bacteriaMatrix.push([]);
                this.antibioticMatrix.push([]);
                this.foodMatrix.push([]);
                for (var j = 0; j < this.h; ++j) {
                        // Matrix cell values less than 0 indicate
                        //   that the cell is vacant.
                        this.bacteriaMatrix[i].push(-1);
                        this.antibioticMatrix[i].push(-1);
                        this.foodMatrix[i].push(-1);
                }
        }
};

/*-- Matrix functions --*/

Environment.prototype.boundX = function (x) {
        return ((x%this.w)+this.w)%this.w; // allow negative numbers
};

Environment.prototype.boundY = function (y) {
        return ((y%this.h)+this.h)%this.h; // allow negative numbers
};

// @description Adds either a bacteria, antibiotic or food
```

```
//   into the environment, adding both to the list and matrix.
Environment.prototype.add = function (toAdd, silent) {
        var type = 'unknown';
        var list = null;
        var matrix = null;
        var x = toAdd.x = this.boundX(toAdd.x);
        var y = toAdd.y = this.boundY(toAdd.y);

        if (toAdd instanceof Bacteria) {
                type = 'bacteria';
                list = this.bacteriaList;
                matrix = this.bacteriaMatrix;
        }
        else if (toAdd instanceof Antibiotic) {
                type = 'antibiotic';
                list = this.antibioticList;
                matrix = this.antibioticMatrix;
        }
        else {
                type = 'food';
                list = this.foodList;
                matrix = this.foodMatrix;
        }

        if (matrix[x][y] < 0 && this.foodMatrix[x][y] < 0) {
                for (var i = 0; i < list.length; ++i) {
                        if (list[i] == null) {
                                list[i] = toAdd;
                                matrix[x][y] = i;
                                ++this[type + 'Count'];
                                return;
                        }
                }
                if (!silent) {
                        console.error('Cannot add ' + type + ': ' +
                                'no list vacancy @ ' + i);
                }
        } else {
                if (!silent) {
                        console.error('Cannot add ' + type + ': ' +
                                'no matrix vacancy @ ' + x + ',' + y);
                }
        }
};

Environment.prototype.remove = function (toRemove) {
        var type = 'unknown';
        var list = null;
        var matrix = null;
        var x = toRemove.x = this.boundX(toRemove.x);
```

```
                var y = toRemove.y = this.boundY(toRemove.y);
                var listIndex = -1;

                if (toRemove instanceof Bacteria) {
                        type = 'bacteria';
                        list = this.bacteriaList;
                        matrix = this.bacteriaMatrix;
                }
                else if (toRemove instanceof Antibiotic) {
                        type = 'antibiotic';
                        list = this.antibioticList;
                        matrix = this.antibioticMatrix;
                }
                else {
                        type = 'food';
                        list = this.foodList;
                        matrix = this.foodMatrix;
                }

                listIndex = matrix[x][y];

                if (listIndex >= 0) {
                        matrix[x][y] = -1;
                        list[listIndex] = null;
                        --this[type + 'Count'];
                }
        };

        // @description Return an array of 0 to 8 neighboring bacteria cells.
        Environment.prototype.getAdjacent = function (cell) {
                var neighbors = {
                        bacteria: [],
                        antibiotic: [],
                        food: []
                };
                for (var i = -1; i < 2; ++i) {
                        for (var j = -1; j < 2; ++j) {
                                if (i != 0 || j != 0) {
                                        var x = this.boundX(cell.x + i);
                                        var y = this.boundY(cell.y + j);
                                        var ai = this.antibioticMatrix[x][y]; // antibioticList
index
                                        var bi = this.bacteriaMatrix[x][y]; // bacteriaList index
                                        var fi = this.foodMatrix[x][y]; // foodList index
                                        if (ai >= 0) {

neighbors.antibiotic.push(this.antibioticList[ai]);
                                        }
                                        if (bi >= 0) {
                                                neighbors.bacteria.push(this.bacteriaList[bi]);
```

```
                                }
                                if (fi >= 0) {
                                        neighbors.food.push(this.foodList[fi]);
                                }
                        }
                }
        }
        return neighbors;
};

// @description Return the coordinates of an empty adjacent
//   Environment cell. If one does not exist, the coords will be (-1, -1).
//   This MUST be done in a random order.
Environment.prototype.getEmptyAdjacent = function (cell) {
        var emptyNeighbors = {
                bacteria: [],
                antibiotic: [],
                food: []
        };
        for (var i = -1; i < 2; ++i) {
                for (var j = -1; j < 2; ++j) {
                        if (i != 0 || j != 0) {
                                var x = this.boundX(cell.x + i);
                                var y = this.boundY(cell.y + j);
                                var ai = this.antibioticMatrix[x][y]; // antibioticList
index
                                var bi = this.bacteriaMatrix[x][y]; // bacteriaList index
                                var fi = this.foodMatrix[x][y]; // foodList index
                                if (ai < 0) {
                                        emptyNeighbors.antibiotic.push({ x: x, y: y });
                                }
                                if (bi < 0) {
                                        emptyNeighbors.bacteria.push({ x: x, y: y });
                                }
                                if (fi < 0) {
                                        emptyNeighbors.food.push({ x: x, y: y });
                                }
                        }
                }
        }
        return emptyNeighbors;
};


/*-- Biological Functions --*/

Environment.prototype.populate = function (bacteriaCount, antibioticCount, dna) {

        var xycoords = []; // array of unique xy-coordinates
        var matchCoordinates = function (a, b) {
```

```
                return a.x == b.x && a.y == b.y;
        };

        bacteriaCount = bacteriaCount || 20;
        antibioticCount = antibioticCount || 10;

        while (xycoords.length < antibioticCount + bacteriaCount) {
                xycoords.pushUnique({
                        x: Math.floor(Math.random()*this.w),
                        y: Math.floor(Math.random()*this.h)
                }, matchCoordinates);
        }

        for (var i = 0; i < bacteriaCount; ++i) {
                this.add(new Bacteria(xycoords[i].x, xycoords[i].y));
        }

        for (var j = i; j-i < antibioticCount; ++j) {
                this.add(new Antibiotic(xycoords[j].x, xycoords[j].y));
        }
};

// @description Spreads antiobitic radially, given that the antibiotic has
//   not reached a minimum density. This is calculated using the number of
//   additional cells occupied by antibiotic each generation.
//   See variable: this.antibioticDiffusion
Environment.prototype.updateAntibiotic = function () {
        var al = this.antibioticList.slice(); // create copy
        var alength = al.length;
        for (var i = 0; i < alength; ++i) {
                if (al[i]) al[i].spread();
        }
        this.resolveChallenges();
};

Environment.prototype.updateBacteria = function () {
        var bl = this.bacteriaList.slice(); // create copy
        var blength = bl.length;
        for (var i = 0; i < blength; ++i) {
                if (bl[i]) bl[i].update();
        }
};

Environment.prototype.updateFood = function () {

        if (generation % settings.food.regenerationRate == 0) {
                environment.add(new Food());
        }

        var fl = this.foodList.slice(); // create copy
```

```
                var flength = fl.length;
                for (var i = 0; i < flength; ++i) {
                        if (fl[i]) fl[i].update();
                }
        };

        Environment.prototype.resolveChallenges = function () {
                for (var i = 0; i < this.w; ++i) {
                        for (var j = 0; j < this.h; ++j) {
                                var ai = this.antibioticMatrix[i][j];
                                var bi = this.bacteriaMatrix[i][j];
                                if (ai > -1 && bi > -1) {
                                        if (this.antibioticList[ai].kills(this.bacteriaList[bi]))
{
                                                this.remove(this.bacteriaList[bi]);
                                        } else {
                                                this.remove(this.antibioticList[ai]);
                                        }
                                }
                        }
                }
        };

})();
```

Main

```
// This is a prototype of the algorithms used in simulating bacteria
// evolution. See python scripts in the "src" directory for final project.
(function () {
        'use strict';

        var Simulation = function () {
                var that = this;
                this.PAUSED = false;
                this.BEGUN = false;
                this.founders = {
                        food: [],
                        bacteria: [],
                        antibiotic: []
                };

                // Globals
                window.environment = new Environment();
                window.generation = 0; // current generation (global)

                canvas.on('click', function (e) {
                        that.canvasClickHandler(e);
                });
```

```
        canvas.render();
        this.report();
};

Simulation.prototype.start = function () {
        // Start button is also the reset button
        if (this.BEGUN === false) {
                this.PAUSED = false;
                this.BEGUN = true;
                this.loop(100);
        } else {
                this.PAUSED = true;
                this.BEGUN = false;
                this.reset();
        }
        flipStartButton(this.BEGUN);
};

Simulation.prototype.stop = function () {
        this.PAUSED = true;
};

Simulation.prototype.step = function () {
        this.PAUSED = false;
        this.BEGUN = true;
        this.loop(100);
        this.PAUSED = true;
        flipStartButton(this.BEGUN);
};

Simulation.prototype.reset = function () {

        // Delete the current environment and reset to defaults
        window.generation = 0;
        delete window.environment;
        window.environment = new Environment();

        // Add the copies back into the environment.
        // Create copies to avoid modification.
        this.founders.food.forEach(function (f) {
                environment.add(new Food(f.x, f.y));
        });
        this.founders.bacteria.forEach(function (b) {
                environment.add(new Bacteria(b.x, b.y));
        });
        this.founders.antibiotic.forEach(function (a) {
                environment.add(new Antibiotic(a.x, a.y));
        });
```

```
        // Re-render
        this.report();
        canvas.render();
        generationTick();
};

Simulation.prototype.report = function () {
        updateCounters();
        console.log('\n\nGeneration ' + generation + '\n');
        console.log('Bacteria count: ' + environment.bacteriaCount);
        console.log('Antibiotic count: ' + environment.antibioticCount);
        console.log('Food count: ' + environment.foodCount);
};

Simulation.prototype.canvasClickHandler = function (e) {
        var pos = getCanvasMousePosition(canvas.canvas, e, true);

        if (settings.simulation.spawnbrush == 'bacteria') {
                environment.add(new Bacteria(pos.x, pos.y), true);
                if (this.BEGUN === false) {
                        this.founders.bacteria.push(new Bacteria(pos.x, pos.y));
                }
        }
        else if (settings.simulation.spawnbrush == 'antibiotic') {
                environment.add(new Antibiotic(pos.x, pos.y), true);
                if (this.BEGUN === false) {
                        this.founders.antibiotic.push(new Antibiotic(pos.x, pos.y));
                }
        }
        else {
                environment.add(new Food(pos.x, pos.y), true);
                if (this.BEGUN === false) {
                        this.founders.food.push(new Food(pos.x, pos.y));
                }
        }

        updateCounters();
        canvas.render();
};

Simulation.prototype.checkSuccess = function () {
        return environment.antibioticCount == 0
                && generation > settings.antibiotic.genx;
};

Simulation.prototype.checkFailure = function () {
        return environment.bacteriaCount == 0;
};

Simulation.prototype.loop = function (speed) {
```

```
                if (this.PAUSED === false) {
                        this.update();
                } else {
                        return;
                }

                // A: Have the cells overcome the antibiotic?
                if (this.checkSuccess() === true) {
                        antibioticDefeated();
                        console.log('\nBacteria are immune after ' + generation + '
generations.');
                }

                // B: Have the cells all died?
                else if (this.checkFailure() === true) {
                        bacteriaDefeated();
                        console.log('\nBacteria are dead after ' + generation + '
generations.');
                }

                // C: Cells are still evolving...
                else {
                        setTimeout(function (that) {
                                that.loop(speed);
                        }, speed, this);
                }
        };

        /*-- Evolution Loop --*/

        Simulation.prototype.update = function () {

                // Next generation
                generation += 1;
                generationTick();

                environment.updateAntibiotic();
                environment.updateBacteria();
                environment.updateFood();

                this.report();
                canvas.render();
        };

        window.sim = new Simulation();
})();
```

Canvas

```
(function () {
	'use strict';

	var Canvas = function (canvas) {
		var that = this;
		this.canvas = canvas;
		this.ctx = this.canvas.getContext('2d');
		this.ctx.textAlign = 'center';
		this.ctx.font = '18px Helvetica';
		this.tilew = Math.floor(this.canvas.width/settings.environment.width);
		this.tileh = Math.floor(this.canvas.width/settings.environment.height);

		this.colors = {
			background: '#002b36',
			text: '#FCFEFB',
			food: '#6c71c4',
			bacteria: '#859900',
			antibiotic: '#dc322f',
			challenge: '#b58900'
		};

		// Indicate the current "highlighted" tile
		this.on('mousemove', function (e) {
			that.onHover(e);

			// Mouse is "painting" (LMB down) -- hack
			if (e.which == 1) {
				window.sim.canvasClickHandler(e);
			}
		});

		this.on('mouseout', function (e) { that.render(); });
	};

	// More attractive event handlers binding
	Canvas.prototype.on = function (eventName, handler) {
		this.canvas.addEventListener(eventName, handler);
	};

	/**
	 * @description Render the entire simulation scene.
	 * @param {Environment} environment
	 * @param {Boolean} DEBUGGING - Should debugging text be rendered?
	*/
	Canvas.prototype.render = function(DEBUGGING) {
		this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
		for (var i = 0; i < environment.w; ++i) {
```

```
                        for (var j = 0; j < environment.h; ++j) {
                                var cx = this.tilew*i;
                                var cy = this.tileh*j;
                                var tx = cx + this.tilew/2;
                                var ty = (cy + this.tileh/2) + 5;
                                var ai = environment.antibioticMatrix[i][j];
                                var bi = environment.bacteriaMatrix[i][j];
                                var fi = environment.foodMatrix[i][j];
                                this.ctx.globalAlpha = 1;

                                // TODO Change PROB(survival) calculations

                                // TODO - isn't there a better way of doing this? Do we care?

                                if (ai >= 0 && bi >= 0) {
                                        var psurvival = 1 -
environment.antibioticList[ai].potency/100;
                                        this.ctx.fillStyle = this.colors.challenge;
                                        this.ctx.fillRect(cx, cy, this.tilew, this.tileh);
                                        if (DEBUGGING) {
                                                this.ctx.fillStyle = this.colors.text;
                                                this.ctx.fillText(psurvival, tx, ty);
                                        }
                                } else if (ai >= 0) {
                                        var potency = environment.antibioticList[ai].potency;
                                        this.ctx.globalAlpha =
potency/settings.antibiotic.potency;
                                        this.ctx.fillStyle = this.colors.antibiotic;
                                        this.ctx.fillRect(cx, cy, this.tilew, this.tileh);
                                        if (DEBUGGING) {
                                                this.ctx.globalAlpha = 1;
                                                this.ctx.fillStyle = this.colors.text;
                                                this.ctx.fillText(potency, tx, ty);
                                        }
                                } else if (bi >= 0) {
                                        var dna = environment.bacteriaList[bi].dna;
                                        this.ctx.fillStyle = this.colors.bacteria;
                                        this.ctx.fillRect(cx, cy, this.tilew, this.tileh);
                                        if (DEBUGGING) {
                                                this.ctx.fillStyle = this.colors.text;
                                                this.ctx.fillText(dna, tx, ty);
                                        }
                                } else if (fi >= 0) {
                                        var sustenance = environment.foodList[fi].sustenance;
                                        this.ctx.fillStyle = this.colors.food;
                                        this.ctx.fillRect(cx, cy, this.tilew, this.tileh);
                                        if (DEBUGGING) {
                                                this.ctx.fillStyle = this.colors.text;
                                                this.ctx.fillText(sustenance, tx, ty);
                                        }
```

```
                        }
                    }
                }
        };

        /**
         * @description Draw a single cell.
        */
        Canvas.prototype.onHover = function (e) {
                var pos = getCanvasMousePosition(canvas.canvas, e, true);
                pos.x *= this.tilew;
                pos.y *= this.tileh;
                this.render(false);
                this.ctx.globalAlpha = 0.6;
                this.ctx.fillStyle = this.colors[settings.simulation.spawnbrush];
                this.ctx.fillRect(pos.x, pos.y, this.tilew, this.tileh);
        };

        window.canvas = new Canvas(document.querySelector('canvas'));
})();
```

Settings

```
(function () {
        'use strict';

        window.settings = {
                // Default Simulation Settings
                simulation: {
                        // Possible values: bacteria, antibiotic, food
                        spawnbrush: 'bacteria'
                },

                // Default Environment Settings
                environment: {
                        width: 20,
                        height: 20
                },

                // Default Antibiotic Settings
                antibiotic: {
                        genx: 5,
                        potency: 150,
                        diffusal: 20
                },

                // Default Bacteria Settings
                bacteria: {
                        energy: 50,
```

```
                thresholds: { lower: 33, upper: 66 },
                priorities: {
                        first: 'mate',
                        second: 'food',
                        third: 'replicate'
                },
                cost: {
                        move: 1,
                        mate: 15,
                        replicate: 35
                },
                mutationRate: 5, // %
                mutationStep: 10
        },

        food: {
                sustenance: 80,
                regenerationRate: 25
        }
};

})();
```

Utilities

```
(function () {
        'use strict';

        // Shuffle array using Knuth Shuffle method
        window.knuthShuffle = function (l) {
                var i = l.length, t, ri;
                while (0 !== i) {
                        ri = Math.floor(Math.random()*i);
                        i -= 1;
                        t = l[i];
                        l[i] = l[ri];
                        l[ri] = t;
                }
                return l;
        };

        window.removeCellsFromList = function (cells, list) {
                if (!(cells instanceof Array)) cells = [cells];

                for (var i = 0; i < cells.length; ++i) {
                        list = list.filter(function (toMatch) {
                                if (toMatch.cid === cells[i].cid) return false;
                                else return true;
                        });
```

```
        }
        return list;
};

/* @description Only push the item if it does not exist
 * already in the array. Duplicates are identified using
 * the comparison function parameter.
 *
 * @param {any} item: The element to be pushed onto the array.
 * @param {function} compare: Function which accepts two items
 *   from the array and returns true when they match, and false
 *   otherwise. The first parameter will always be the item which
 *   is being added to the array.
 *
 * @returns {Boolean}
 *   true: The item was added to the array.
 *   false: The item was NOT added to the array.
*/
Array.prototype.pushUnique = function (item, compare) {
    if (typeof item == undefined) return false;
    if (typeof compare != 'function') {
        console.error('Invalid comparison function');
        return false;
    }
    for (var i = 0; i < this.length; ++i) {
        if (compare(item, this[i]) === true) return false;
    }
    this.push(item);
    return true;
};

/* @description Create a new array containing all the elements
 * similar between the two arrays. Similar elements are identified
 * using the comparison function parameter.
 *
 * @param {Array} array: The Array object with which to intersect.
 * @param {Function} compare: Function which accepts two elements,
 *   one from each array. It must return true when they are similar,
 *   and false otherwise.
 *
 * @returns {Boolean}
 *   true: The item was added to the array.
 *   false: The item was NOT added to the array.
*/
Array.prototype.intersect = function (array, compare) {
        if (typeof array == undefined) return [];
        if (typeof compare != 'function') {
                console.error('Invalid comparison function');
                return [];
        }
```

```
        var intersect = [];
        var a = this.length < array.length ? this : array;
        var b = this.length < array.length ? array : this;
        for (var i = 0; i < a.length; ++i) {
            for (var j = 0; j < b.length; ++j) {
                if (compare(a[i], b[j]) == true) {
                    intersect.push(a[i]);
                }
            }
        }
        return intersect;
    };

    /**
     * @description Get the HTML5 canvas-relative xy-coordinates
     *   of a mouse. This function assumes there is an mouse-X event
     *   object to be provided. If not, fake it..
     * @param {Canvas Object} c: The canvas in which the mouse is acting.
     * @param {DOM Event} e: The event object created by the mouse action.
     * @param {Boolean} normalize: Whether or not to convert the coordinates
     *   into canvas tile size-relative coordinates.
     * @returns {Object} Integer pair (x,y).
    */
    window.getCanvasMousePosition = function (c, e, normalize) {
           var b = c.getBoundingClientRect(); // canvas bounds
           var tilew = Math.floor(c.width/settings.environment.width);
           var tileh = Math.floor(c.width/settings.environment.height);
           var pos = {
                  x: Math.round((e.clientX-b.left)/(b.right-b.left)*c.width),
                  y: Math.round((e.clientY-b.top)/(b.bottom-b.top)*c.height)
           };

           if (!normalize) return pos;
           else return {
                  x: Math.floor(pos.x / tilew),
                  y: Math.floor(pos.y / tileh)
           };
    };

})();
```