Project 3

PROJECT 3

In this project, we continue working with 2D simulations. This week we are using agent-based simulations and linked-lists to control out 2D simulations. I copied over our Cell and Landscape classes from last week and made changes to both. I also created two new classes called CategorizedCell and Simulation. For this project, Cells keep track of their own location and Landscapes are linked-lists that keep track of the agents on the Landscape. Overall, this project tested and improved my knowledge of LinkedLists,Java arrays, ArrayLists, for loops, for each loops, double-nested for loops, conditionals, and some new interesting methods.

Task 1. This task was fairly straightforward. I was asked to make the Cell class keep track of its own location with the double fields of x and y. We had to create a few new simple methods, a toString that returns a period, and an empty updateState method[to be edited later].

Task 2. In this task, I was asked to update my Landscape class to give it width and height parameters as well as a LinkedList of Cell objects. Again, we had to create new methods within this class. One method, addAgent simply added a Cell to the landscape.The three most complex methods in this task were the toString(), the getNeighbors() and the advance(). First, toString() creates a string representation of the Landscape. To do this, we had to create an array of empty String objects and then go through each agent and concatenate all of the Strings in the array and return it. We also had to make sure ALL agents were within the boundaries of the Landscape. I received help from TA Mike with this method. Here is a snippet of what it looks like:

```
public String toString() {
    String[][] slist = new String[this.getRows()][this.getCols()];
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            slist[i][j]  = " " ;
        }
    }
    for(Cell x: this.getAgents()){
        if((x.getRow() < height && x.getRow() >=0) && (x.getCol() < width && x.getCol() >=0)) {
            slist[x.getRow()][x.getCol()] = x.toString();
        }
    }

    String finalStr = new String();
    for( int i = 0; i < slist.length; i++){
        for(int j= 0; j<slist[i].length; j++){
            finalStr += slist[i][j];
        }
        finalStr += "\n";
    }
    return finalStr;
}
```

Next, I created a getNeighbors function that returns an ArrayList of all the Cells within a given radius of the current location. I used the distance formula and Math.pow() to calculate this. In addition, I created an advance method that called updateState on a shuffled list of cells. Here is a snippet of the code for those two methods:

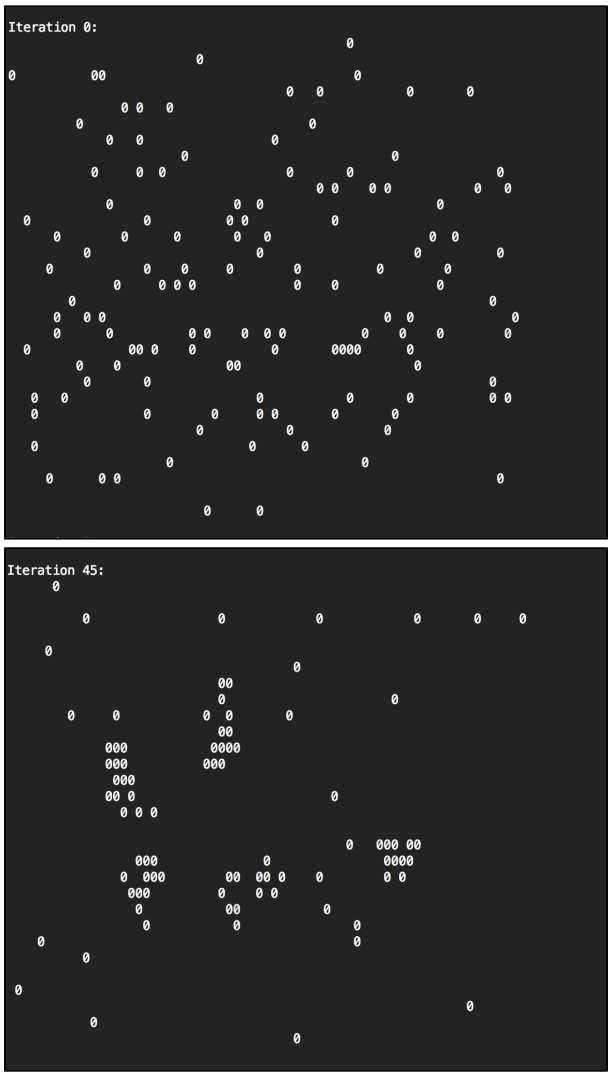
```
   public ArrayList<Cell> getNeighbors(double x0, double y0, double radius){
       ArrayList<Cell> neighbors = new ArrayList<Cell>();
       for(Cell z : landscape){
           if(Math.pow(x0 - z.getX(), 2) + Math.pow(y0 - z.getY(), 2) <= (radius * radius)){
               neighbors.add(z);
           }
       }
       return neighbors;
   }


   public void advance(){
       landscape.toShuffledList();
       for(Cell x : landscape){
           x.updateState(getNeighbors(x.getX(), x.getY(), 2));
       }
   }
```

Task 3. In this task, we had to write the updateState method that we previously left blank in the Cell class. It had to follow specific rules. If a cell had more than 3 neighbors, move +/- 5 with a 1% chance and else move +/- 5. To do this, I first created a method called randomInRange() that generated a random value between two given numbers. In this case, -5 and 5! I received help from Steve Parrot, his tutor, and StackOverflow on this method. Then, I created the updateState method that satisfied all of the rules. Here is a snippet of that code:

```
 //Help and collaboration with Steve Parrot
 public void updateState(ArrayList<Cell> neighbors) {
     int numNeigh = neighbors.size();
     Random rand = new Random();
     if (numNeigh > 3) {
         int percentage = rand.nextInt(100);
         if (percentage == 0) {
             this.x0 += randomInRange(-5, 5);
             this.y0 += randomInRange(-5, 5);
         }
     }
     else {
             this.x0 += randomInRange(-5, 5);
             this.y0 += randomInRange(-5, 5);
     }
 }
```

The behavior of this Landscape of agents should result in a clumping behavior. When I run 45 iterations, clumping can easily be seen. Here are

two pictures from Iteration 0 and Iteration 45:



Task 4. Now, in this task, I create a new class called CategorizedCell that gives each Cell one more field that is a category of cell. In this project, at first, a CategorizedCell could be one of two categories: 0 or 1. I had to write a new toString and updateState method. The toString returned a string representation of the category. So, if the Cell was category 1, it would return a "1". Next, the updateState method was a little different. It had different rules, like if there are more cells with the same category than there are cells of a different category, move +/- 5 with a 1% chance. To do this, I created counters and found number of same and different CategorizedCells. Here is a snippet of the code:

```
public void updateState(ArrayList<Cell> neighbors){
    int catSame = 0;
    int catDiff = 0;
    for( Cell c : neighbors){
        if( c instanceof CategorizedCell){
            //help from CP Majgaard--forced type cast
            if( ((CategorizedCell)c).getCategory() == this.getCategory()){
                catSame ++;
            }
            else{
                catDiff++;
            }
        }
        //this is if the cell doesn't have a category
        else{
            catDiff ++;
        }
    }
```

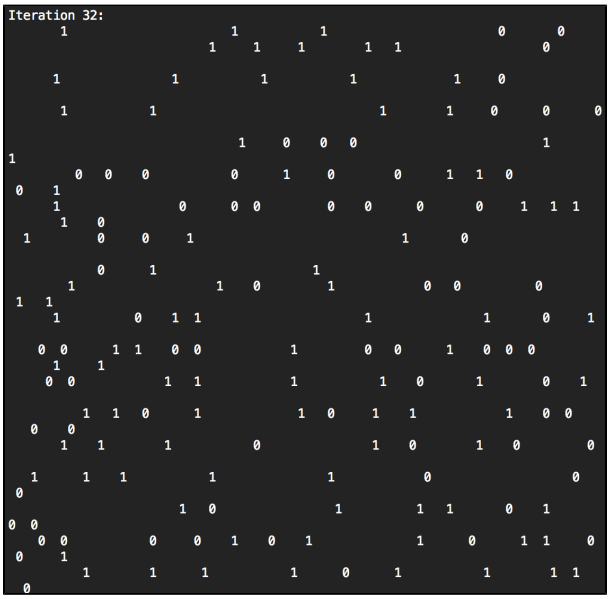
```
    Random rand = new Random();
    if (catSame > catDiff) {
        int percentage = rand.nextInt(100);
        if (percentage == 0) {
            this.x0 += randomInRange(-5, 5);
            this.y0 += randomInRange(-5, 5);
        }
    }
    else {
            this.x0 += randomInRange(-5, 5);
            this.y0 += randomInRange(-5, 5);
    }
}
```

Task 5. Here we edited the main test function in the Landscape class and used command line arguments to determined which type of Cell is created: either a normal Cell or a CategorizedCell. To do this, I coded that if the user typed 1 command line argument it would create a Cell and if the user typed 2 args it would return a CategorizedCell. Professor Maxwell helped me here. Here is a snippet:

```
  if( args.length != 1 && args.length != 2){
      System.out.println("You need either 1 or 2 command line args");
      System.out.println("If you want a CatCell use 2 args");
      System.out.println("If you want a normal Cell use 1 arg");
      return;
  }
  //Help from Bruce on command line args TASK 5
  for(int i=0; i < N; i++) {
      if(args.length == 1){
          scape.addAgent( new Cell(
                      gen.nextFloat() * (cols-1) ,
                      gen.nextFloat() * (rows-1)) );
      }
      else if(args.length == 2){
          scape.addAgent( new CategorizedCell(
                      gen.nextFloat() * (cols-1) ,
                      gen.nextFloat() * (rows-1), (int) gen.nextInt(2) ));
      }
  }
```

So, when I run Landscape with one arg, for example java Landscape Cell, I will run the normal Cell and get something like this:

And, if I run Landscape with two args, for example java Landscape CatCell 2, I will run the CategorizedCell and get something like this:



Task 6. This task asked me to simply create a Simulation class that can run either type of simulation and controls the iterations, how often the simulation is printed, the rows and columns, and the agents. During this task, I completed **extension1**. I made everything command line arguments. Basically, I created a new main function that had command line args. The first one was type. If type was = 0, a new Cell was created and if type was = 1 and new CategorizedCell was created. Here is a snippet of that:

```
for(int i=0; i < agents; i++) {
    if(type == 0){
        scape.addAgent( new Cell(
                    gen.nextFloat() * (cols-1) ,
                    gen.nextFloat() * (rows-1)) );
    }
    //Comment for task 7
    else if(type == 1){
        scape.addAgent( new CategorizedCell(
                    gen.nextFloat() * (cols-1) ,
                    gen.nextFloat() * (rows-1), (int) gen.nextInt(2) ));
    }
```

Task 7. This task called for another new type of agent that has a different update rule. I created a rule that if the category was 2, then the cells would un-clump. What I did was, if the number of same categories was less than the number of differnet categories thatn I would +/- 5 20% of the time. This caused the type "2" agents to move away from the others. CP helped me here. Here is a snippet of that code:

```
//Task 7--if the category value = 2
//This category type will unclump! instead of clumping
if(this.getCategory() == 2){
    Random rand = new Random();
    if (catSame < catDiff) {
        int percentage = rand.nextInt(100);
        if (percentage >= 79) {
            this.x0 += randomInRange(-5, 5);
            this.y0 += randomInRange(-5, 5);
        }
    }
    else {
            this.x0 += randomInRange(-5, 5);
            this.y0 += randomInRange(-5, 5);
    }
}
else{
```

We also had to edit the toString method to account for the new category that returns "2" when the category is 2.

Extension 1. I mentioned this extension in task 6. I decided to make all of my simulation class controlled by command line args. First, I created a condition to break the execution if there weren't enough args. That looked like this:

```
public static void main(String[] args) {
    if( args.length != 6){
        System.out.println("You need 6 command line arguments!");
        System.out.println("The command line arguments should follow this format:");
        System.out.println("java Simulation type iters rows cols agents print");
        System.out.println("IMPORTANT:");
        System.out.println("Input 0 for type Cell, Input 1 for type CategorizedCell");
        return;
    }
```

Then, I set everything I needed equal to a specific arg[]. That looked like this:

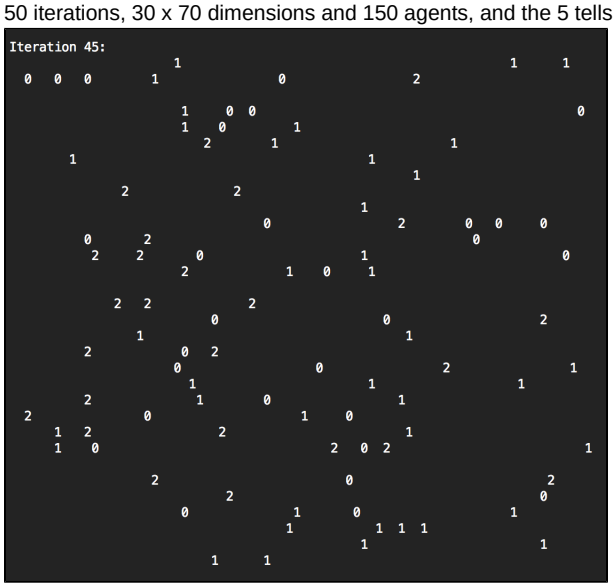
```
int type = Integer.parseInt(args[0]);
int iters = Integer.parseInt(args[1]);
int rows = Integer.parseInt(args[2]);
int cols = Integer.parseInt(args[3]);
int agents = Integer.parseInt(args[4]);
int print = Integer.parseInt(args[5]);
Landscape scape = new Landscape(cols, rows);
Random gen = new Random();
```

To determine how many iterations to print out, I used a modulo. CP helped me code this:

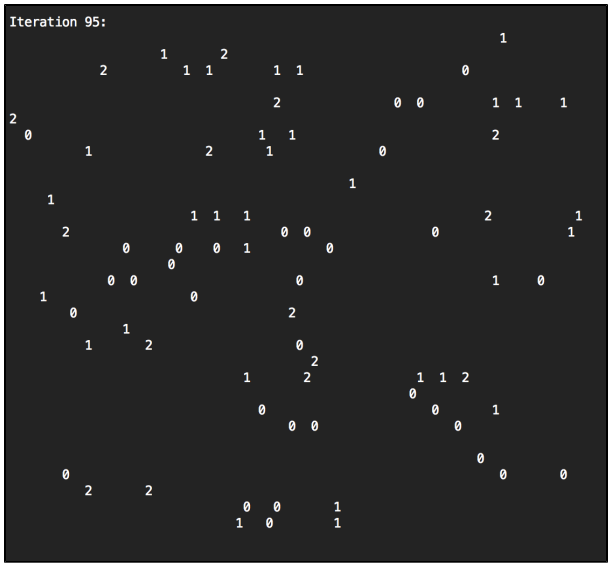
```
for(int i=0; i < iters; i++) {
    scape.advance();
    //Help from CP. Use modulo to determine how many iterations to print out
    //For example, 10 % 5(print) leaves no remainder
    //So that will print every 5 iterations
    if(i == 0 || i % print == 0){
        System.out.printf("Iteration %d:\n", i);
        System.out.println( scape );
    }
}
```

So, if I typed a command line like this into the terminal -- java Simulation 0 50 30 70 150 5 -- that creates a Landscape of Cells with 50 iterations, 30 x 70 dimensions and 150 agents, and the 5 tells it to print every 5 iterations. Here is what will print out:

If I then did the same thing but changed the first arg to 1, (java Simulation 1 50 30 70 150 5)-- that creates a Landscape of Categorized Cells with 50 iterations, 30 x 70 dimensions and 150 agents, and the 5 tells it to print every 5 iterations. Here is what will print out:



Extension 2. The second extension addressed mixing and matching different agent types. I completed part of this in task 7. Instead of only putting 2 types of categorized cells on the landscape (ie. 0&1 or 0&2), I added all three agent types at once, showing their behavior. What should be seen is that 0's and 1's clump separately, while 2's avoid all clumping. So, I ran all three types(0,1,2) at once a observed their behavior. When I run the command line: java Simulation 1 100 30 70 150 5 I get this:



You can begin to see how the 0's and 1's clump together and the 2's un-clump.

Conclusion. I feel like i really learned a lot in this project. It took a lot of time and question asking to get it right. I really am starting to understand the Java language better. I improved my grasp on 2D arrays and Landscape. I learned how to implement linked-lists to control 2D simulations. In the end, the biggest thing I learned is how to create 2D Landscape simulations with different Cell agents, with different rules, and even different categories.

Who helped me. I received help from Bruce Maxwell, CP Majgaard, TA Mike, and worked alongside CS 231 student Steven Parrot throughout the project.