Criterion C: Product Development

Techniques used

- Weighted/Dynamic probability
    - Random number generation according to a distribution
    - Biology-inspired algorithm
- Lists (Arraylists java) as dynamic data structures
- Iterators for looping data structures and random number generation
    - Concurrency (for on-the-fly modification)
    - Modify the dynamic data structures indirectly
- Binary search with nearest value
- Bidimensional arrays as data structures
    - Array manipulation
- Object oriented programming
    - Variables/Attributes
    - Functions/Methods
    - Class inheritance
- Graphical representation
    - Array scaling
    - Alpha value modification
        - Bit shifting
- Multiple Variables

First and foremost, I should mention that the program quickly grew in complexity, as noted by the list of used techniques. Therefore, it is not possible due to word constraints to explain every single one of them, so I opted for outlining the most important ones and deeply explaining only the ones that I consider most relevant.

For simulating the behavior of ants, four bidimensional arrays will be used, which represent the maps of the pheromones distribution, one for the pheromone used to search for food, and another to return home, these also include the obstacles that stop ants from advancing; the other arrays represent the ant's position, and the location of the nest and the food. This maps are implemented as classes and provide methods for setting values in a range in the array, using nested loops; constructors for setting up the bidimensional according to certain parameters (“width” and “height”). The Map class, also provides methods using during the debugging and testing phases, these methods basically print or set the values from arrays in different ways. Most, interestingly, the Maps class contains a method for modyfing an array held by each ant which specifies the pheromone distribution in the ant's neighbourhood.

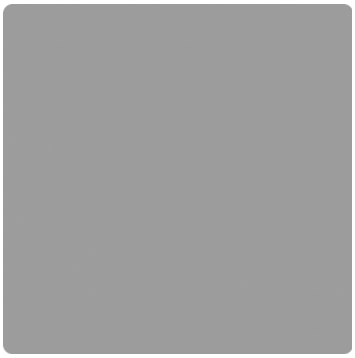
Finally, the Map class has a “child class” called PherMap which inherits the characteristics from its parent's class, but adds certain methods and variables of its own, such as minimum and maximim values per cell, and the pheromone diffusion method, which lowers the value of each cell as time passes, so that unefficient paths are less prioritized.

Initially, I considered using a single pheromone, however it has been suggested that a two-pheromone system provides a more significant representation of the actual way ants forage, I found this idea first from a simulation project done by Kevin Workman and also took some inspiration from Forrest O's work. The Ant class provides the methods for choosing the direction to move, and interacts with methods from other classes such as Maps.

The ants will be stored in an ArrayList in the Simulation class, because such structure provides the required methods for a dynamic data structure, which is essential for simulating birth and death of ants without stopping the simulation and manually recreating the array.

This processes will be managed by the Simulation class, which provides the necessary methods for updating the status of the simulation and displaying it graphically.

Fundamentally, an ant will be represented a black square in the simulation grid, and is to be placed on its proportional X,Y location according to the window size. Ants will be allowed to overlap within the same grid, up to a certain extent, proportional to the total number of ants. This will be graphically represented as having higher or lower levels of alpha values or transparency.



Square with some ants

Saturated square (full with ants)

In a similar fashion, the pheromone level in each cell will be represented as varying levels of blue/red for the food and home pheromones respectively by transparency and the mixture of both. In other terms, as the pheromone level in a certain cell gets higher, so does its opaqueness, and if a certain cell contains both values, then the rgb values will be added so as to *combine* the colors. Because Processing stores colors in a 24 bit integer, this is done via bitwise operators –right shifting and OR masking– (for higher performancev) in the following way:

```
color c = (alpha<<24) | (red_value<<16) | (green_value<<16) |
(blue_value<<8);
```

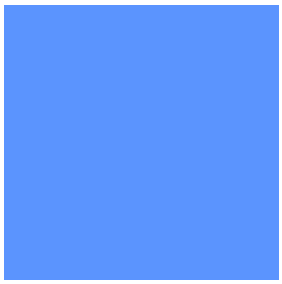
The alpha value is calculated according to a scale of the minimum and maximum value of the pheromone array (1 and 100 respectively) and the ones from the alpha values (0 to 255). Processing's map function allows us to nicely scale values according to initial and final minimum values and maximum values.

```
 alpha = (int)map(foodpher.pointMap[rows][cols],1,500, 1,255);
```

Example of Alpha coloring:

Light red pheromone (for food)

Intense red pheromone



Light blue pheromone (for home)

Intense blue pheromones

Cell with both pheromones

The core of the program is the next cell calculation method, which probabilistically calculates the next move to be made according to the pheromone map in the ant's locality. The code for the main ant decision calculation is outlined below (exploded to include methods from other classes or other methods within the same class), real variable names are not used for the sake of the explanation. This functionality is divided into four methods from the Ant class, and an aditional one from the Map class which updates the ant locality array. Basically, what the code does is scan the eight cells surrounding the ant, get their pheromone value, then generate an array of a size that represents the sum of the probabilities time a number (the “granularity” or “precision”), fill it with the values of its neighbours times the precision number, and randomly select from such array. Finally, the algorithm gets the index from such value in the original probability array, and updates the ant's location according to the result from the calculation.

We will proceed to explain this algorithm by commenting on a simplified version of the code (excluding similar branches) and providing a simple example that serves as an illustrative process:

```
//In the simulation class, inside a loop for each ant (represented as variable a)
  updateAntNeigh(a); // Calls method from simulation class
  a.move(); //Calls move method from Ant class
```

```
//Outside of the loop; another method. This one checks if the ant is
     //hungry or not and calls the appropiate method from the Maps
     //class. Code if not_hungry excluded

  void updateAntNeigh(Ant a) {

      if (a.is_hungry) { //a.is_hungry is a booleean attribute from
                         //the Ant class

        foodpher.getNeighbours(a); //Method from Maps class called on
                                   // foodpher instance, with Ant a
                                   // as parameter

    }
```

For understanding the reasoning behind the following part of the code, the next table is essential, which illustrates the index the ants neighbourhood array and their x,y coordinates with respect to the ant's. From now on and for illustrative purposes we will assume the probability of each cell, is its index plus one, this is denoted by the dummy header Value.

Index: 6 (x-1,y-1) Value: 7	7 (x,y-1) Value: 8	Index: 0 (x+1,y-1) Value: 1
Index: 5 (x-1,y) Value: 6	ANT (x,y)	Index: 1 (x+1,y) Value: 2
Index: 4 (x-1,y+1) Value: 5	Index: 3 (x,y+1) Value: 4	Index: 2 (x+1,y-1) Value: 3

```
//Code from Map class

  void getNeighbours(Ant a){

    int y = a.locationy; //Get coordinate attributes from ant

    int x = a.locationx;

    float answer[] = new float[8]; //Create temporary array of size 8
                                   //because of the neighbourhood

    setAllsimple(-1, answer); //Presets every value to -1 because if
```

```
                                    //unmodified, then it is an obstacle
    if (y-1>=0+1 &&   x+1 <gridsizew-1  && a.visited[y-1][x+1] != 1
      ){ answer[0] = pointMap[y-1][x+1]; //The location should not
                                         //exceed the grid's size and
                                         // should not be visited in
                                         //order to count the full
                                         // value
  }else if (y-1>=0+1 &&   x+1 <gridsizew-1 && a.visited[y-1][x+1] ==
1){ answer[0] = 0.15;}                   //If it has been visited,
                                         //regardless of its value it
                                         // gets a penalty and is
                                         //reduced to .15, so that
                                         //looping is discouraged


    if (x+1 <gridsizew-1 && a.visited[y][x+1] != 1) {answer[1] =
pointMap[y][x+1]*1.5;                    //If the index is odd, and not
                                        // diagonal, it gets a 50%
                                        //bonus, as diagonal movements
                                        //should be more expensive
                                        //given the higher distance
                                        //they cover
ETC. . . And so on until all 8 cells are covered.


    a.prob=answer; //Modifyies the array probability attribute of the
                   //ant
  }
```

The most important part of the algorithm consists of the actual probabilistical calculations, these are contained in three methods from the Ant class:

This code will be called from the siulation class, for each existing ant in each simulation cycle,

followed by the displacement of said ant, and the placement of the appropriate pheromone in the new cell.

```
 void evaluateDirections(){ //Called from move method
  int tmpsize = 8;
  for (float m: prob){ //Calculate sum of all probabilities by a
                       //for each loop in Java
                       //Prob is the array containing the pheromone
                       //values in the ant's neighbourhood
    if (m <=0) {
      tmpsize--; //If an obstacle (-1) is found, then the size of the
                 // array decreases, so do the posssible indexes
    }else{
      total+=m;
    }
  }
  randomDir(createRandom(prob, total), prob); //Call two methods
                                              //below in a nested
                                              //fashion


  }
float[] createRandom(float[] p,float oavg){
  int size = (int)(oavg*100)+1;  //Precision is 100, so the size of
                                 //the array is the sum times a
                                 //hundred, as each number will be
                                 //added its value times 100 to the
                                 //array.
  float arr[] = new float[size];
```

```
int index = 0;                  //The index of the arr array
for (int x = 0; x<8;x++){ //Nested for loop, for each value in
                          //original array
  for(int y=0; y<(p[x]*100) && index<size; y++, index++){
                                //This for loop has two conditions and
                                //incrementing values, the first of
                                //them refers to each value's times to
                                //be added, and the second one, to the
                                //total possible elements in the array.
                                //Two indexes were implemented because
                                //rounding errors sometimes led to
                                //attempts that tried to access
                                //unexisting elements in the array,
                                //resulting in a crash
    if (p[x] >0) arr[index]= p[x]; //If it is a valid number and
                                   //not an obstacle
  }
}
return arr;                     //The method returns the calculated
                                //array which hold the distribution of
                                //the random numbers
```

Out sample array would have the following distribution

Value	Number of appearancces
8	800
7	700
6	600
5	500
4	400
3	300
2	200

1	100

```
}
int randomDir(float[]p, float[]o){ //This method gets a random number
                                   //from our crafted distribution
                                   //and finds its index in the
                                   //original array
  int r = (int)random(0,p.length);
  float x = p[r];
   if (p[r] ==0) return r; //If no possibilities exist,then follow a
                           //random path
  int index;
  ArrayList<Integer> poss = new ArrayList<Integer>();
 //This dinamic array holds possible indexes, in case two are the
//same.

  for(index =0; index<8;index++){
    if(Math.abs(o[index] - x) < 0.001) { //As the array contains a
                                         //float, we cannot perform an
                                         //equality test using '==',
                                         //therefore, we are forced to
                                         /choose an epsilon, and
                                         //verify that the abosule
                                         //value between both is
                                         //within an epsilon of
                                         //distance
          poss.add(index);
    }
```

```
  }


  next = poss.get(rn.nextInt(poss.size())); //Random value from our
                                            //index array

 return index;

}
```

Then we, perform the movement according to the indexes in our original table, after the movement is performed, a pheromone is planted in the corresponding cell, and it is marked as visited in a local (of each Ant object) array.

Another algorithm worth mentioning is the modified binary search which scans a sorted array and instead of returning the exact position of a value, it returns its position within the existing values of the array, displacing each value to its smaller neighbour. This is used for converting the pixel location of the mouse, to our simulated grid's size by comparing coordinates.

Word Count 1460