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
    - Concurrent modification
- Binary search modified for arrays
- Bidimensional arrays as data structures
    - Array manipulation
- Object oriented programming
    - Functions
    - Class inheritance
- Graphical representation
    - Coordinate system transformation
    - Alpha value modification
        - Bit shifting
- Multiple Variables

As I developed the program, it 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 less important ones and deeply explaining only the ones that I consider most relevant.

For simulating the behavior of ants, bidimensional arrays will be used, which represent the pheromone distributions, 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. These maps are implemented as classes and provide methods for modifying and printing arrays using nested loops; constructors for setting up the bidimensional according to certain parameters ("width" and "height"). There is also a method for modifying an array held by each ant which describes the ant's surroundings.

Finally, the Map class has a "child class" called PherMap which inherits the characteristics from its parent's class, adding certain methods and variables of its own, such as minimum and maximum values per cell, and the pheromone diffusion & evaporation method, which lowers the cells' value pheromone value as time passes and distributes a percentage of it among its neighbors.

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 (Panait et al).

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

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

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.

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 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 performance) in the following way:

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

```
(blue_value<<8);
```

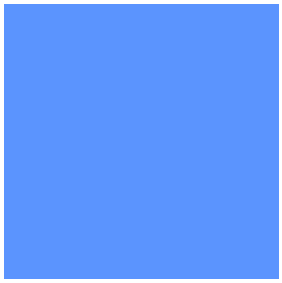
The alpha value is calculated according to a scale of the minimum and maximum value of the pheromone array (1 and 100 respectively) and the ones from the alpha values (0 to 255). Processing's map function allows us to scale values according to initial and final minimum values and maximum values.

```
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 “next cell” calculation method probabilistically calculates an ant's movement according to the pheromone map in its locality. The code for the main ant decision calculation is outlined below, with some modifications for the sake of explanation.

Basically, what the code does is scan the eight cells surrounding the ant, get their pheromone value, then generate an array filled with numbers representing the intervals of probability of each possible

decision, and randomly select a number from 0 to the largest. Finally, the algorithm gets the index from such value in the original 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(); +



//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 ant neighborhood array and their x,y coordinates with respect to the ant's and is filled with dummy values.

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
                                         //cycling 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 following code is what actually calculates the ant's displacement, it is from the corrected version of the algorithm, as the previous one used repetition of numbers instead of intervals, and was much slower than this second version.

```
 void evaluateDirections(){ //Called from move method
  randomDir(createRandom(neighborProbabilities)); //Call two methods
                                                  //below in a nested
double arr1[] = new double[8];
     double accumulator = 0; //This value will serve as a "bookmark"
                             //that stores the summation of all
                             //probabilities in order to create the
                             //intervals
     for (int x = 0; x<8;x++){
          if (p[x] >0.15){   //We need to check that it is not
                             //an obstacle or a previously visited
                             //location
          arr1[x] = accumulator+ Math.pow(reward + p[x], exponent);
                             //These last lines generate the
                             //interval for each pheromone,
                             //by using exponential functions we can
                             //control the pheromone affinity.
                             //This was loosely inspired by more
                             //complicated algorithms used for
```

```
                              //searching (called Ant colony
                              //optimisation)
                    accumulator = arr1[x];
                    lastXbigger = x; //We need to store this as we will
                                     //use it later as an upper limit for
                                     //generating the random number
               }else{
                    p[x] =-1;
                    }
               }
               return arr1;
  }
```

Out sample array would have the following distribution

Value	Upper limit of interval (assume exponent is 2)
8	204
7	140
6	91
5	55
4	30
3	14
2	5
1	1

```
}
int randomDir(double[]o){ //This method gets a randoom value
                          //from our distribution
                          //and returns its index to indicate the next
                          //position of the ant
```

```
double r =  (double) (1+ Math.random() * (o[lastXbigger]-1));
```

//This generates a number ranging from one, to the largest value in the interval

```
 for (int x = 0; x<8; x++){ //We need to cycle the intervals array
                            //until we find a value bigger than the
                            //randomly generated one, which would
                            //mean that it “fell” into the interval
     if (o[x] <= 0) continue;
     else if (o[x] >= r && o[x] >= 0){
          return x;         //X is the index of the next poisition
     }
     }
}
```

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. Two methods with different drawbacks are provided, one performs a relative addition inspired by *Panait Liviu et al*, and the other performs a direct addition. The first one provides faster convergence with the danger of creating loops while the direct additions takes more time to find an optimal path.

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, corresponding each the coordinates of the mouse closest existing “cell”. This is used for converting the pixel location of the mouse, to our simulated grid's size by transforming coordinate systems. This type of algorithm was used because of its efficiency in searching sorted arrays, which at the worst case is of O(log n) where 'n' is the number of items.

Word Count 1259

References:

Liviu Panait and Sean Luke. 2004. Ant Foraging Revisited. In Proceedings of the Ninth International

Conference on the Simulation and Synthesis of Living Systems (ALIFE9), pages 569-574.