On the warring of ants: Modelling competitive ant colonization through random walks

Henrik Åhl
in collaboration with Denhanh Huynh

October 4, 2015

Department of Astronomy and Theoretical Physics, Lund University

Project supervised by Tobias Ambjörnsson

LUND UNIVERSITY

1 Introduction

How animals and insects move is seldom truly something stochastic, although random walks can in some cases be used to mimic an overall behaviour that can be meaningful for larger populations, where the inherent structure is lost in noise. [2] Indeed, also the use of stochastical methods on ant colonies have been done before, with respect to the complex underlying network therein [3]. In this report we instead investigate the dynamics of a system where the competitve nature between different ant breeds play into account; two types of ants entitled `speedAnts` and `bruteAnts` are set out within a grid-based forest to gather food, build up new anthills and make sure to survive battles with ants of different types. The both types within our model are characterized by different values in the the two attributes *speed* and *strength* corresponding to their respective names.

Several collective stochastical processes play deeply into the dynamics of the system. In particular, the walks of the ants are determined by a random direction, biased in the direction of food sources if the ant is not already carrying food, or strongly biased towards friendly anthills if not.

2 Creating a model for colonization

2.1 Problem description

We do in this report try to determine several things with respect to the dynamics of our model. Mainly we want to investigate how the attributes play into the evolution of the system with time. Secondly, as a direct consequence, it is of interest too see whether there can be established local equilibriums, where both ants stagnate in development. Due to the high chaotic nature of the model it is unlikely that a final equilibrium will be reached, why the occurence of local ones is of greater interest. Also the forest size should affect which ant type is or is not at a loss, and how the system develops.

2.2 Description of the model

In essence, the model created to simulate the dynamics is centered around five steps, summarized as follows:

1. Add new food to grid.
2. Let ants of different types combat when standing on the same square.
3. Move ants to an adjacent square, or stay within the initial one.
4. Let ants who stand on squares with food collect.
5. Update anthills and leave off food. If food supply is big enough, create a new anthill. Also generate new ants.

As a practice for our simulations, the number of anthills that could be present within the forest was set at a maximum level to avoid diverging dynamics.

2.3 Where to go: How ants walk

Ant walk biases are determined by the speed inherent to their type, as well as a summation over their different biases – food supplies for ants not carrying food, and friendly anthills for ants with food. These biases are weighed with respect to the inverse of their distance to the ant squared, so that the ant more securely walks in the correct direction the closer it gets to its target. All ants do, however, have a chance not to move during the time step, but rather spend time within its present square. This too is scaled in relation to the speed of the ant type, so that a quicker type will be less likely to stand still during its turn as an effect. The details of this procedure can be found in appendix A under the method `move`.

2.4 Brawling in nature: How ants fight

Whenever two or more ants of opposing sides happen to stand upon the same square, combat will commence. The result of combat is drawn from two non-negative Gaussian distributions, where each distribution corresponds to how many ants of the opposing side the ants manage to kill. In essence, whenever a value < 0 is drawn, it is substituted for a new one. The mean is centered around the sum of the total strength of an ant type, whereas the standard deviation is the root of this, making it likely that the number of ants killed by a species is roughly the same as their cumulative strength. For further reference, code is included in appendix A under the method `combat`.

2.5 Simulation settings

Food is distributed randomly across the grid with a 1 % chance. In total up to three stacks of food will be placed per occurence of this sort, although it is still possible for more food to appear before the other supplies have vanquished. The amount of food is drawn from a uniform distribution, where the range stretches between 0–15000 units.

All anthills generate one new ant per time step. Whenever the supply of nutrition in a separate hill has reached over 8000 units, the hill will migrate to a nearby neighborhood, where the specific position is determined by a uniform distribution. Furthermore, a ceiling was set at 25 anthills in total throughout the forest in order to limit the system numerically.

3 Results and conclusions

It is clear from the results that the matrix size, i.e. how big the modelized forest happens to be, is detrimental to the results of the dynamics throughout a simulation. It is clear that `bruteAnts` are favored by smaller matrices, where interactions are more regular and the competitiveness for the food sources is stronger. `speedAnts` are in turn favored by the larger maps, where interactions are sparse and development is largely determined by your ability to collect food.

As can be seen in fig. 1 and fig. 2, the system is largely determined by the size of the matrix. A larger grid allows for a smoother development, even though it is expected of the system to converge against a one-type scenario, where only one ant type de facto survives. Somewhere around the range of a 25×25 matrix there seems to be a turning point where `speedAnts` are more favored by the environment, and `bruteAnts` fall behind.

Notably, with respect to combat, in the sudden decrease in ants for both types in fig. 2, `speedAnts` make up for their lack in strength through greater numbers, making larger combats fairly equal. The reader should take note howsoever, that this fact is purely inherent to the settings chosen when simulating, although it still shows the effect of allowing for numbers and flexibility to counter pure force in a dynamic system such as this.

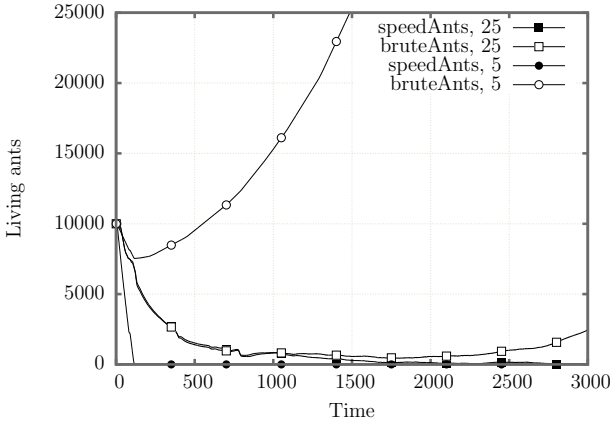

Figure 1: Comparison of dynamics between matrix with side 5 and side 25.

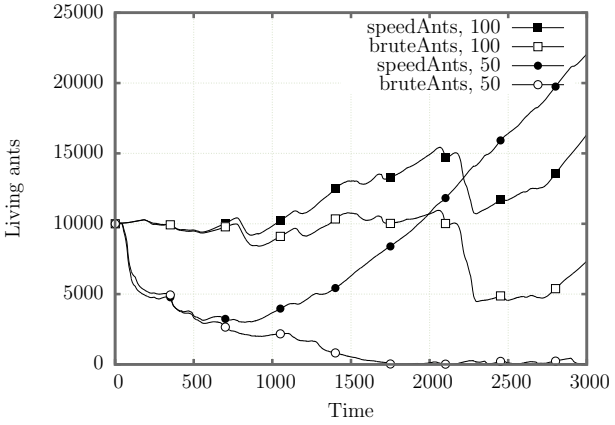

Figure 2: Comparison between matrix with side 50 and side 100

In order to verify the accuracy of the model, the graphical representation illustrated in fig. 3 below figured as an important measure. Accordingly to the graphical output, no typical anomalies could be seen, which enhances the belief that the model indeed did behave as intended.

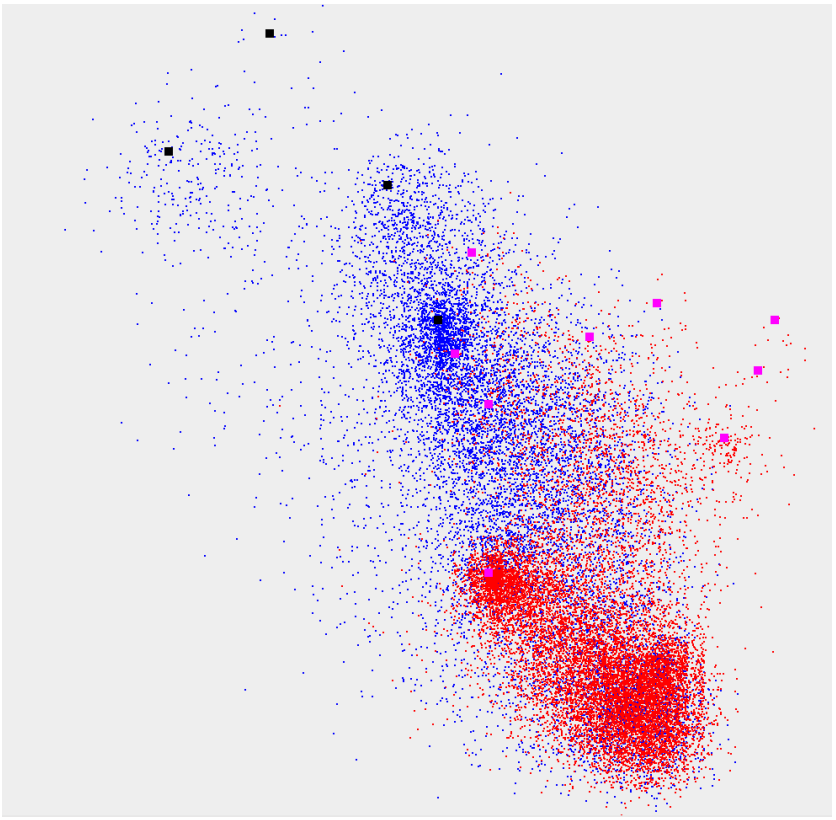

Figure 3: Snapshot of simulation in a 50×50 matrix. `speedAnts` are represented with blue dots (black anthills), while `bruteAnts` are seen as red (magenta anthills). In this capture, ants are seen having just finished a source of food, and are returning homeward.

Contrary to the initial aim of investigation, there seems to be some possibilities of establishing local equilibriums, where both types of ants can live on for an extended amount of time. Especially the 50×50 matrix simulation visualized in fig. 2 displays this behaviour.

However, also the amount of ants with which the system is first initialized with, as well as the amount of food spawned (and in how many places) can gravely shift the stability of how the system develops.

In summary, the dynamics of such a highly stochastic system are hard to properly measure and compare. The severe fluctuations that are inherent to the model, and the demanding computations that forelie makes statistical checks within the thermodynamical limit practically impossible. The system may act as a model for how stochastical principles can determine the nature of such a system as this, although it can not be deemed to be of any practical importance.

References

[1] Tobias Ambjörnsson, *Lecture notes, Computational Physics*, Department of Astronomy and Theoretical Physics, Lund University, 2015.

[2] A. Attanasi et al, *Collective Behaviour without Collective Order in Wild Swarms of Midges*, PLoS Computational Biology: 10(7), 2014.

[3] D. Jin et al, *Ant Colony Optimization with a new Random Walk Model for Community Detection in Complex Networks*, Jilin University, 2013.

A Code

```
/* Gathering class for settings and utilities */
public class Common
{
   public static final int ITERATIONS = 3000;
   // Starting ants per race
   public static final int ANTHILLANTS = 1000000;
   // New ants when migration occurs
   public static final int NEWANTHILLPOP = 0;
   // Root of number of squares
   public static final int MATRIXSIZE = 100;
   // Pixels in frame
   public static final int FRAMESIZE = 1000;
   // Size of a separate square in graphical rep.
   public static final int SQSIZE = FRAMESIZE / MATRIXSIZE;
   // Distance constant for how new anthills to occur
   public static final int ANTHILLDIST = 40;
   // Number of new foodstacks
   public static final int FOODSTACKS = 3;
   // New food size max value
   public static final int FOODSTACKSIZE = 15000;
   // Food limit for when to commence migrationary measures
   public static final int MIGRATE_LIMIT = 8000;
   // Maximum amount of anthills
   public static final int ANTHILL_LIMIT = 25;
   // New ants in anthill per timestep
   public static final int ANTGENERATION = 1;
   // Bias to go towards food
   public static final double FOODBIAS = 3.;
   // Bias to return home when having gathered food
   public static final double HOMEBIAS = 10000.;
   // Combat modifying constant
   public static final double COMBATCONST = 0.001;
   // Chance of new food during timestep
   public static final double FOODCHANCE = 1 / 100.;
   // Constant for chance of standing still
   public static final double NOTHING_DIR = 10.;
}
```

```
import java.util.ArrayList;
import java.util.Iterator;
```

```
import java.util.Random;
import javax.swing.JFrame;

/* Code for simulating competitive ant colonization throughout a
   modeled
 * forest. Ants can be part of either of two species, each with
    different
 * values in the likelihood to move and the potential to kill
    enemy ants. */
public class Forest
{
   private static Square[][] squares;

   private static ArrayList<Food> food = new ArrayList<Food>();
   private static ArrayList<Anthill> anthills = new
      ArrayList<Anthill>();

   public static void main(String[] args)
   {
      Forest forr = new Forest();
      forr.initialize();
      int iterations = Common.ITERATIONS;
      JFrame frame = new JFrame();

      // Sets frame resolution and other parameters.
      frame.setSize(Common.FRAMESIZE, Common.FRAMESIZE);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      AntPanel panel = new AntPanel(frame.getWidth(),
        frame.getHeight());
      frame.getContentPane().add(panel);
      frame.setVisible(true);
      frame.setResizable(false);

      // Run, forest -- run!
      while (iterations-- > 0)
      {
         panel.update(squares, food, anthills);
         frame.getContentPane().validate();
         frame.getContentPane().repaint();
         int spHill = 0, brHill = 0;

         for (Anthill hill : anthills)
            if (hill.getType() == SpeedAnt.INDEX)
```

```
            spHill++;
         else
            brHill++;

      System.out.print(Common.ITERATIONS - iterations + "\t" +
         spHill + "\t"
            + brHill + "\t");
      forr.timeStep();
   }
}

private Random rand = new Random();

// Randomize number of stacks, position and amount of food
public void addFood()
{
   for (int i = 0; i < 1 + rand.nextInt(Common.FOODSTACKS); i++)
   {
      int x = rand.nextInt(squares.length);
      int y = rand.nextInt(squares.length);

      // Check if position already occupied
      for (Food f : food)
      {
         if (f.getX() == x && f.getY() == y)
         {
            f.setAmt(f.getAmt() +
               rand.nextInt(Common.FOODSTACKSIZE));
            return;
         }
      }
      food.add(new Food(x, y,
         rand.nextInt(Common.FOODSTACKSIZE)));
   }
}

public void dropFood(Anthill hill)
{
   // Number of ants with food that should be dropped
   int x = hill.getX();
   int y = hill.getY();

   int amt = squares[x][y].getAnts()[hill.getType()][1];
```

```
   // Subtracts amt ants from ants with food
   squares[x][y].addAnts(hill.getType(), 1, -amt);
   // Adds amt ants to ants without food
   squares[x][y].addAnts(hill.getType(), 0, amt);

   hill.addFood(amt);
}

// Check all food occupied squares and distribute food amongst
   present ants
public void getFood()
{
   Iterator<Food> i = food.iterator(); // Avoid stupid
      concurrency errors
   while (i.hasNext())
   {
      Food f = i.next();
      f.setAmt(squares[f.getX()][f.getY()].getFood(f.getAmt()));
      if (f.getAmt() < 1)
         i.remove();
   }
}

// Initialize Forest
public void initialize()
{
   squares = new Square[Common.MATRIXSIZE][Common.MATRIXSIZE];
   for (int i = 0; i < squares.length; i++)
      for (int j = 0; j < squares[0].length; j++)
         squares[i][j] = new Square();

   // Initialize anthills in opposing quadrants
   anthills.add(new Anthill(
         Common.MATRIXSIZE / 2 + rand.nextInt(Common.MATRIXSIZE
            / 2),
         Common.MATRIXSIZE / 2 + rand.nextInt(Common.MATRIXSIZE
            / 2),
         BruteAnt.INDEX));
   anthills.add(new Anthill(rand.nextInt(Common.MATRIXSIZE / 2),
         rand.nextInt(Common.MATRIXSIZE / 2), SpeedAnt.INDEX));

   // Add starting ants to anthill
```

```
   for (Anthill hill : anthills)

                squares[hill.getX()][hill.getY()].anthillInit(hill.getType());

   // Must start with some food, or we get silly dynamics
   addFood();
}

// Loop through squares and move ants in directions udlr
public void moveAnts()
{
   // Movement mega matrix. Needed not to take multiple steps
     for some ants
   // when traversing matrix.
   int[][][][][] transfer = new
     int[squares.length][squares.length][4][2][2];

   // Calculates how the ants in all squares should move
   for (int i = 0; i < squares.length; i++)
     for (int j = 0; j < squares[0].length; j++)
     {
        Square sq = squares[i][j];
        transfer[i][j] = sq.move(food, anthills, i, j);
     }

   // Moves the ants
   for (int i = 0; i < squares.length; i++)
     for (int j = 0; j < squares[0].length; j++)
     {
        Square sq = squares[i][j];

        // Move ants while accounting for boundaries
        if (j != squares.length - 1)
           sq.transferAnts(squares[i][j + 1],
             transfer[i][j][0]);
        if (j != 0)
           sq.transferAnts(squares[i][j - 1],
             transfer[i][j][1]);
        if (i != 0)
           sq.transferAnts(squares[i - 1][j],
             transfer[i][j][2]);
        if (i != squares[0].length - 1)
           sq.transferAnts(squares[i + 1][j],
```

```
                transfer[i][j][3]);
        }
    }

    // Take a step in time
    public void timeStep()
    {
        int totSp = 0;
        int totBr = 0;

        try
        {
            Thread.sleep(0); // Add sleep time for graphics if you
                want
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        // Add/increment stacks of food by given chance
        if (rand.nextDouble() < Common.FOODCHANCE)
            addFood();

        // Resolve combats due to ants of different types on same
            squares
        for (int i = 0; i < squares.length; i++)
            for (int j = 0; j < squares[0].length; j++)
            {
                totSp += squares[i][j].sumAnts(0);
                totBr += squares[i][j].sumAnts(1);
                squares[i][j].combat();
            }
        System.out.println(totSp + "\t" + totBr);
        moveAnts();
        getFood();
        updateAnthills();
    }

    // Migrate anthills if enough food
    public void updateAnthills()
    {
        ArrayList<Anthill> temp = new ArrayList<Anthill>();
        for (Anthill hill : anthills)
```

```
      {
         // Leave off food at home
         dropFood(hill);
         // Generate new ants
         squares[hill.getX()][hill.getY()].addAnts(hill.getType(),
           0,
              Common.ANTGENERATION);

         // If enough food and not too many anthills already
         if (hill.getFood() > Common.MIGRATE_LIMIT
               && anthills.size() <= Common.ANTHILL_LIMIT)
         {
            int x, y;
            // Randomize new location
            while (true)
            {
               x = hill.getX() - Common.ANTHILLDIST / 2
                     + rand.nextInt(Common.ANTHILLDIST);
               y = hill.getY() - Common.ANTHILLDIST / 2
                     + rand.nextInt(Common.ANTHILLDIST);
               if ((x >= 0 && x < squares.length) && y >= 0
                     && y < squares[0].length)
                  break;

            }
            // Add new anthill to list
            temp.add(new Anthill(x, y, hill.getType()));
            // Remove food from hill
            hill.nullFood();
            // Add new ants to anhill
            squares[x][y].addAnts(hill.getType(), 0,
              Common.NEWANTHILLPOP);
         }
      }
      anthills.addAll(temp);
   }

}
```

```
import java.util.ArrayList;
import java.util.Random;

public class Square
```

```
{
   // Row: type
   // Column: no food / has food
   private int[][] ants = new int[2][2];
   Random rand = new Random();

   // Initialize square with zero ants
   public Square()
   {
      for (int i = 0; i < ants.length; i++)
         for (int j = 0; j < ants[i].length; j++)
            ants[i][j] = 0;
   }

   // Initialize square with ant matrix
   public Square(int[][] ants)
   {
      this.ants = ants;
   }

   public void addAnts(int i, int j, int amt)
   {
      ants[i][j] += amt;
   }

   public void anthillInit(int type)
   {
      ants[type][0] = Common.ANTHILLANTS;
   }

   // Let ants not carrying food on a select square fight
   public void combat()
   {
      int[] totalAntStr = { ants[0][0] * SpeedAnt.STR,
            ants[1][0] * BruteAnt.STR };

      if (totalAntStr[0] > 0 && totalAntStr[1] > 0)
      {
         double[] kills = { -1, -1 };

         // Take number of kills from non-negative normal
            distribution
         for (int j = 0; j < kills.length; j++)
```

```
            while (kills[j] < 0) // Only take positive values
               kills[j] = Common.COMBATCONST
                     * (rand.nextGaussian() *
                        Math.sqrt(totalAntStr[j])
                           + totalAntStr[j]);

         // Kill ants w/ and wo/ food fractionally. Food is lost
            upon kill.
         for (int i = 0; i < ants.length; i++)
         {
            int temp = ants[i][0] + ants[i][1];

            for (int j = 0; j < ants[i].length; j++)
            {
               ants[i][j] -= kills[(i + 1) % 2] * ants[i][j] / temp;

               // No negative ants plz
               if (ants[i][j] < 0)
                  ants[i][j] = 0;
               assert(ants[i][j] >= 0);

            }
         }
      }
   }

   public int[][] getAnts()
   {
      return ants;
   }

   // Gives out food to unfed ants proportionally, and returns new
      food stock at
   // location.
   public int getFood(int food)
   {
      int unfed = ants[0][0] + ants[1][0];
      if (unfed == 0)
         return food;
      if (food >= unfed)
      {
         ants[0][1] += ants[0][0];
         ants[0][0] = 0;
```

```
      ants[1][1] += ants[1][0];
      ants[1][0] = 0;
      return food - unfed;
   }

   // Categorizes ants according to available food.
   ants[0][1] += ants[0][0] * food / unfed;
   ants[0][0] -= ants[0][0] * food / unfed;
   ants[1][1] += ants[1][0] * food / unfed;
   ants[1][0] -= ants[1][0] * food / unfed;
   return 0; // No food left. :(
}

// Returns number of travelers of different groups, where first
   column is
// square in direction u/d/l/r. Calculates inclination to walk
   in certain
// direction based on positions of food (if the ants don't
   already carry any)
// or friendly anthills (if the ants carry food). Distance is
   based on a
// Pythagorean measure.
public int[][][] move(ArrayList<Food> food, ArrayList<Anthill>
   anthills,
      int x, int y)
{
   int[][][] directions = new int[4][2][2]; // Return matrix
   double speed;

   for (int i = 0; i < ants.length; i++)
   {
      if (i == SpeedAnt.INDEX)
         speed = SpeedAnt.SPEED;
      else
         speed = BruteAnt.SPEED;
      for (int j = 0; j < ants[i].length; j++)
      {
         for (int l = 0; l < ants[i][j]; l++)
         {
            // The higher the value of the direction, the more
               likely the ant
            // is to move in that direction
            double udlrn[] = { speed, speed, speed, speed,
```

```
      Common.NOTHING_DIR / speed }; // Move dirs
ArrayList list = (j == 0) ? food : anthills;

for (Object obj : list)
{
   double rx, ry;
   if (list == anthills)
   {
      if (((Anthill) obj).getType() == i)
      {
         rx = ((Anthill) obj).getX() - x;
         ry = ((Anthill) obj).getY() - y;
      } else
         continue;
   } else
   {
      rx = ((Food) obj).getX() - x;
      ry = ((Food) obj).getY() - y;
   }

   // Move according to whether we have food or not
   if (rx * rx + ry * ry != 0)
   {
      double add = list == anthills ? speed *
         Common.HOMEBIAS
            / Math.sqrt((rx * rx + ry * ry) * (rx *
               rx + ry * ry)
                  * (rx * rx + ry * ry))
            : speed * Common.FOODBIAS * ((Food)
               obj).getAmt()
                  / Math.sqrt(
                        (rx * rx + ry * ry) * (rx * rx
                           + ry * ry)
                              * (rx * rx + ry * ry));

      // Add direction bias
      if (ry > 0)
         udlrn[0] += add * ry;
      else if (ry < 0)
         udlrn[1] += add * -ry;
      if (rx < 0)
         udlrn[2] += add * -rx;
      else if (rx > 0)
```

```
                udlrn[3] += add * rx;
            }
        }

        // Don't go out of limits
        if (y == Common.MATRIXSIZE - 1)
            udlrn[0] = 0;
        if (y == 0)
            udlrn[1] = 0;
        if (x == 0)
            udlrn[2] = 0;
        if (x == Common.MATRIXSIZE - 1)
            udlrn[3] = 0;

        // Sum for normalizing
        double sum = udlrn[0] + udlrn[1] + udlrn[2] +
            udlrn[3]
                + udlrn[4];

        // Normalize walk bias
        for (int k = 0; k < udlrn.length - 1; k++)
            udlrn[k] /= sum;

        double R = rand.nextDouble();

        // Randomize direction!
        double chanceSum = 0;
        for (int k = 0; k < udlrn.length - 1; k++)
        {
            chanceSum += udlrn[k];
            if (R < chanceSum)
            {
                directions[k][i][j]++;
                break;
            }
        }
    }
  }
 }
 return directions;
}

// Summarize total number of ants of a type in a square
```

```
   public int sumAnts(int type)
   {
      int temp = 0;
      for (int j = 0; j < ants[type].length; j++)
         temp += ants[type][j];
      return temp;
   }

   // Transfers ants from this square to another.
   public void transferAnts(Square sq, int[][] tr)
   {
      for (int i = 0; i < ants.length; i++)
         for (int j = 0; j < ants[i].length; j++)
         {
            ants[i][j] -= tr[i][j];
            assert(ants[i][j] >= 0);
            sq.addAnts(i, j, tr[i][j]);
         }
   }
}
```

```
public class Food
{
   private int x, y, amount;

   public Food(int x, int y, int amount)
   {
      this.x = x;
      this.y = y;
      this.amount = amount;
   }

   public int getAmt()
   {
      return amount;
   }

   public int getX()
   {
      return x;
   }

   public int getY()
```

```
   {
      return y;
   }

   public void setAmt(int amt)
   {
      this.amount = amt;
   }
}
```

```
public class Anthill
{
   private int x;
   private int y;
   private int food = 0;
   private int type;

   public Anthill(int x, int y, int type)
   {
      this.x = x;
      this.y = y;
      this.type = type;
   }

   public void addFood(int amt)
   {
      food += amt;
   }

   public int getFood()
   {
      return food;
   }

   public int getType()
   {
      return type;
   }

   public int getX()
   {
      return x;
   }
```

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

   // Erase food in anthill (due to migration)
   public void nullFood()
   {
      food = 0;
   }

}
```

```
import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;
import java.util.Random;

import javax.swing.JPanel;

public class AntPanel extends JPanel
{
   // private Image screenBuffer;
   Random rand = new Random();
   Square[][] squares;
   private ArrayList<Food> food;
   private ArrayList<Anthill> anthills;

   public AntPanel(final int width, final int height)
   {
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);

      // Draw all ants
      for (int i = 0; i < squares.length; i++)
         for (int j = 0; j < squares.length; j++)
         {
            int[][] ants = squares[i][j].getAnts();
```

```
      // Draw first ant type
      g.setColor(Color.BLUE);
      for (int k = 0; k < ants[0][0] + ants[0][1]; k++)
         g.fillOval(Common.SQSIZE * i +
            rand.nextInt(Common.SQSIZE),
               Common.SQSIZE * j +
                  rand.nextInt(Common.SQSIZE), 3, 3);

      // Draw second ant type
      g.setColor(Color.RED);
      for (int k = 0; k < ants[1][0] + ants[1][1]; k++)
         g.fillOval(Common.SQSIZE * i +
            rand.nextInt(Common.SQSIZE),
               Common.SQSIZE * j +
                  rand.nextInt(Common.SQSIZE), 3, 3);
   }

// Draw all food
g.setColor(Color.GREEN);
for (int i = 0; i < food.size(); i++)
{
   Food f = food.get(i); // Stupid errors if for-each ...
   int radius = (int) Math.sqrt(f.getAmt());
   g.fillOval((int) (f.getX() * Common.SQSIZE),
         (int) (f.getY() * Common.SQSIZE), (int)
           Math.sqrt(f.getAmt()),
         radius);
}

// Draw all anthills
for (int i = 0; i < anthills.size(); i++)
{
   Anthill hill = anthills.get(i); // Stupid errors if
     for-each ...

   if (hill.getType() == 0)
      g.setColor(Color.BLACK);
   else
      g.setColor(Color.MAGENTA);

   g.fillRect((int) (hill.getX() * Common.SQSIZE),
         (int) (hill.getY() * Common.SQSIZE), 10, 10);
}
```

```
   }

   public void update(Square[][] squares, ArrayList<Food> food,
         ArrayList<Anthill> anthills)
   {
      this.squares = squares;
      this.food = food;
      this.anthills = anthills;
   }
}
```

```
public abstract class Ant
{

}
```

```
import java.util.ArrayList;

public class SpeedAnt extends Ant
{
   static final int INDEX = 0; // index for keeping track of the
     ant races
   static final int STR = 1;
   static final int SPEED = 10;
}
```

```
public class BruteAnt extends Ant
{
   static final int INDEX = 1; // index for keeping track of the
     ant races
   static final int STR = 10;
   static final int SPEED = 2;
}
```