Evolver.java

```
package org.bm.bmtron;

import org.jgap.*;
import org.jgap.gp.*;
import org.jgap.gp.impl.*;
import org.jgap.gp.function.*;
import org.jgap.gp.terminal.*;

import java.io.*;

/**
* This is the main class that defines the problem to be evaluated and sets up
* the population to be evolved to meet that problem. Important methods: main
* sets up the configuration, instantiates an Evolver, and handles evolution.
* create() sets up the command base. fitnessCheck runs an individual in the
* population in a few games to test its mettle.
*
* @author Richard Banister
* @version 1.0
*/
public class Evolver extends org.jgap.gp.GPProblem {

      public Evolver(GPConfiguration config) throws InvalidConfigurationException {
            super(config);
      }

      /**
      * Evaluate fitness of a chromosome. See interior for how exactly this is
      * measured. But generally, the program is placed in a game grid and tested
      * against computer players. 5 games are run and the average score is taken
      * and returned. There is probably some overhead in creation of games that
      * should be reduced, for time's sake. It still takes a while (10s of
      * millis) to run a game; it would be nice to get this down further.
      *
      * @return a double indicated how good the program is. Currently
      * this will be in the range of 0.0 to 3.0.
      * @variable tron is the program to be evaluated.
      */
      public static double fitnessCheck(IGPProgram tron){
            //INVISIBLE BIKE!!

            /*
            * Let's do a few trials so we get a meaningful score, and not just a
            * lucky win or two.
            * Like, let's try five for now
            */
            int trials = 5;
            double runningScore = 0.0;
```

```
Game game;

for (int i = 0; i < trials ; i++){
      //start a new game. Is there a better way to do this, so as to
      //avoid incurring too much overhead?
      game = new Game();

  Player[] players = new Player[2];

      //players[0] = new ComputerPlayer1(new Field(game), "SARK", java.awt.Color.blue, 1);
      players[0] = new ComputerPlayer2(new Field(game), "MCP", java.awt.Color.blue, 1);
      if (tron!=null) players[1] = new GPPlayer(new Field(game), "RAM ", java.awt.Color.red, 0, tron);
      else players[1] = new GPPlayer(new Field(game), "RAM ", java.awt.Color.red, 0);

      game.setPlayers(players);

      game.init(null, null);

      try{
            game.loader.join();
      }catch(java.lang.InterruptedException e){}

      //numRounds is how long the game lasted (in rounds, duh). max==875
      //If you're wondering, it's a double so we calculateify it later
      double numRounds = (new Integer(game.field.getRounds())).doubleValue();

      Player winnar = game.field.getWinner();
      if (winnar!=null){ //may be null, maybe if there's a tie?
            if (winnar.getName().equals("RAM ")){
                  //WE WON WE WON WE WON!!!
                  runningScore+=1.0;

                  /* now, how did we win?
                     add points depending on how opponent crashes:
                       - boundary wall +0.0
                       - their own wall +0.25
                       - our wall +1.0 */
                  Player k = game.field.getKiller();
                  if (k!=null) { //null indicates player hit a bounday wall
                        if (!k.getName().equals("RAM ")) runningScore+=0.25;
                        else runningScore+=1.0;
                  } //else: no points for you!

                  //shorter is better
                  runningScore += 1.0 - (numRounds / 875);

                  //highest possible score for winning: almost 3-ish

            }else{
                  //*sigh* you can't win 'em all.
```

```
                        //give points for how long we lasted - longer is better
                        //BUT, we don't want to give too many points, i don't think
                        runningScore += numRounds/875;

                        //highest possible score for losing: 1.0
                  }
            }else{ //winnar is null
                  //Let's assume that means we tied. That's good enough for +0.5,
                  //don't you think?
                  runningScore+=0.5;
                  //what about game time? I guess longer is better for a tie
                  runningScore += numRounds/875;

                  //highest possible score for tieing[sic]: 1.5
            }
      }

      //check fitness
      double fitness = runningScore/trials;

      return fitness;
}

public static void main(String args[]){
      Evolver evo = null;
      double mi = 0;
      double fitness;

      //Don't do this anymore (it was just to test overhead)
      for (int i = 0; i <  0; i++){
            try { evo = new Evolver(null);}catch(Exception e){}
            mi = System.currentTimeMillis();
            fitness = evo.fitnessCheck(null);

            System.out.println("fitness: "+fitness+"\n\n");
            System.out.println("runtime: "+(System.currentTimeMillis()-mi));
      }

      GPConfiguration config = null;
      try{
            config = new GPConfiguration();
            config.setGPFitnessEvaluator(new DefaultGPFitnessEvaluator());

            config.setMaxInitDepth(12);         //orig:
            config.setPopulationSize(25); //small for now

            config.setFitnessFunction(new fitnessCycle());

            //I still don't really know what this stuff does,
            //besides break everything.
```

```
  /*config.setStrictProgramCreation(false);
  config.setProgramCreationMaxTries(30);

  config.setMaxCrossoverDepth(5);*/

  //Taken from anttrail. WORTH INVESTIGATING.
  config.setCrossoverProb(0.9f);        //orig: 0.9f
  config.setReproductionProb(0.1f); //orig: 0.1f
  config.setNewChromsPercent(0.3f); //orig: 0.3f
  config.setStrictProgramCreation(true);
  config.setUseProgramCache(true);

  config.setCrossoverMethod(new BranchTypingCross(config));

  evo = new Evolver(config);
  GPGenotype gp = evo.create(); //create() is sugoi taisetsu. mitene.

gp.setVerboseOutput(true);

  //Print the whole initial population out
  int popSize = gp.getGPPopulation().getPopSize();
  for (int i = 0; i < popSize; i++){
        gp.outputSolution(gp.getGPPopulation().getGPProgram(i));
  }

  int f = 0; //f will be our breakpoint, if we don't reach out goal
  double fit = 0.0; //current fitness score
  double lfit = 0.0;      //largest fitness score
  IGPProgram prog = null; //for saving the best chromosome
  while (fit < 2.5){
        //old:
        //fitnessCycle.computeFitness(gp.getGPPopulation().determineFittestProgram(), true) < 2.5

        //new:
        System.out.print("Evolving round "+f+". Fitness: ");
        gp.evolve();
        fit = gp.getFittestProgram().getFitnessValue();
        System.out.println(fit);

        if (fit >= lfit) {
              prog = gp.getFittestProgram();
              lfit = fit;
        }

        if (++f == 1000) break;
  }
  //old: just evolve a lot then take the bestest
//gp.evolve(30);
//gp.outputSolution(gp.getAllTimeBest());

  //new: use the bestest one that we maintained in the loop above
```

```
            gp.outputSolution(prog);

            //fitnessCycle.computeFitness(gp.getAllTimeBest(), true);
          //evo.showTree(gp.getAllTimeBest(), "evo_best.png");
          evo.showTree(prog, "evo_best.png");

            /* Try saving it to a binary file.
               This is what BMTron will load for Computer 3 in graphical mode
               I tried saving the whole population but that didn't work out at
               all. But it'd be nice to be able to save a population then load
               it and continue evolving, right? So you could, for example,
               run an applet and test members of the population against actual
               human players. Well, let's just call that TODO
            */
            try{
                  File fSol = new File("evolver_genotype.obj");
                  FileOutputStream fos = new FileOutputStream(fSol);
                  ObjectOutputStream obj_out = new ObjectOutputStream(fos);

                  //old:
                  /*IGPProgram best = gp.getAllTimeBest();
                  best.setApplicationData(null);
                  obj_out.writeObject(best);*/

                  prog.setApplicationData(null);//prevents serialization problems
                  obj_out.writeObject(prog);

                  obj_out.close();
                  fos.close();
            }catch(Exception e){
                  System.out.println("failed to save for some reason:");
                  e.printStackTrace();
            }

      }catch(Exception e){
            e.printStackTrace();
      }


}

/**
* Create makes an initial population. The specification of commands and
* inputs and such is done here, and then a _random_ population is created.
*/
public GPGenotype create() throws InvalidConfigurationException {
    GPConfiguration conf = getGPConfiguration();

    int[] minDepths = new int[] {6};
    int[] maxDepths = new int[] {20};
```

```
  /*
  * I'll talk a little about the CommandGene stuff here, I guess. types[]
  * contains the RETURN types of each nodeSet. A nodeSet being a program
  * block. You should be able to connect nodeSets together somehow, but
  * so far I'm still unclear about this - both the method and the reasons
  * for doing so. Probably has something to do with the return types. And
  * argTypes. argTypes[][] is a list of argument types each nodeSet
  * expects. These get passed into the nodeSets as terminals
  * automatically.
  *
  * Note there are no arguments, no terminals, and the return type is
  * void. This is best way I found of actually getting the programs to
  * do anything. Mixing types didn't seem to work - it always just wanted
  * to return a terminal and wouldn't do nothin' else. Further
  * investigation certainly wouldn't hurt. Well, it might hurt a little.
  */

Class[] types = { CommandGene.VoidClass };

Class[][] argTypes = { {  } };

  //Commands: Classes means "types they expect to work on", NOT return types. I THINK.

CommandGene[][] nodeSets = {
        {
          new SubProgram(conf, new Class[] {
              CommandGene.VoidClass,
              CommandGene.VoidClass,
              CommandGene.VoidClass
          }),
        new SubProgram(conf, new Class[] {
              CommandGene.VoidClass,
          CommandGene.VoidClass,
          CommandGene.VoidClass,
          CommandGene.VoidClass
              }),
              new GPIfWallAheadThen(conf),
              new GPIfWallInFive(conf),
              new GPTurnRight(conf),
              new GPTurnLeft(conf),
              new GPRandomMove(conf),
              new GPFollowOpponent(conf),
              //new GPGoStraight(conf),
              new GPMoveTowardsOpp(conf),
              new GPMoveAwayFromOpp(conf),
        },
};


return GPGenotype.randomInitialGenotype(conf, types, argTypes, nodeSets,
minDepths, maxDepths, 5000, new boolean[] {true}, true);
```

```
            //return GPGenotype.randomInitialGenotype(conf, types, argTypes, nodeSets, 200, true);
      }

      /**
      * Measures fitness. Actually calls fitnessCheck inside Evolver, for
      * convenience sake. (did you get the pun? fitness cycle? hahahaha!
      * whatever)
      *
      * @see GPFitnessFunction
      */
      public static class fitnessCycle extends GPFitnessFunction {
            protected double evaluate(org.jgap.gp.IGPProgram a_subject){
                  return fitnessCheck(a_subject);
            }

            /**
            * Currently unused. alternate fitness checker instead of static one in Evolver
            */
            public static double computeFitness(IGPProgram a_subject, boolean debug){
                  //unused
                  return 1.0;
            }
      }

}
```

GPPlayer.java

```
/*
 * Okay, this'll be my attempt at creating some kind of learning agent.
 * first step is to just create a player that does _something_
 * Note: to set this up, we overwrote ComputerPlayer3 in PlayerSetup.
 * There's a section that dynamically loads the Player classes depending
 * on what was selected... I cut out CP3 (which was apparently unfinished,
 * and wasn't loading by default) and threw this in. The other change that
 * needs to be made is maxControl in PlayerSetup, I think
 */
package org.bm.bmtron;

import java.awt.*;
import java.util.*;
import java.io.*;

import org.jgap.gp.*;

public class GPPlayer extends ThreadedPlayer implements Serializable {
      int length = 1; //temp.
      int turnCount = 0;
      int pos = 0;
```

```
  int lastDirection = Field.UP;

  IGPProgram theProgram = null;

GPPlayer(Field field, String name, Color color, int team) {
    super(field, name, color, team);
}

GPPlayer(Field field, String name, Color color, int team, IGPProgram prog) {
    this(field, name, color, team);
        theProgram = prog;
}

  //This is a super major hack. Not supposed to be able to instantiate w/o
  //real info. This is because for some reason JGAP is trying to serialize
  //this as application data, even though i don't want it to and told it not
  //to. So, solve that and then remove this.
  GPPlayer(){}

  public void init(){
        super.init();
        setDirection(field.UP);

        /*
        * If we don't have a routine yet, we'll load it from disk. This happens
        * when we run in graphical mode. When evolving, the program will always
        * be passed to us. Which is why I feel comfortable outputting text here
        */
        if (theProgram == null){
              System.out.println("BLOAD TRON");
              try{
                    File fSol = new File("evolver_genotype.obj");
                    FileInputStream fis = new FileInputStream(fSol);
                    ObjectInputStream obj_in = new ObjectInputStream(fis);
                    theProgram = (IGPProgram) obj_in.readObject();

                    //let's see what the program looks like:
                    System.out.println(theProgram.toStringNorm(0));

                    obj_in.close();
                    fis.close();

              }catch(Exception e){
                    System.out.println("loading problem:");
                    e.printStackTrace();
              }
        }

        //if we failed loading above, this will cause a big problem. But then,
        //it's nothing but problems from here so why handle it elegantly?
```

```
        theProgram.setApplicationData(this);

  }

synchronized void start() {
        init();
    super.start();
}

  /**
  * Make a moving decision. Go through some procedure, then set the direction
  * to up, down, left or right. The GPPlayer now loads the chromosome, which
  * contains the moving instructions, and queries it for it decision.
  */
protected void move() {
        /* The first temporary AI for testing purposes. It forms a left-turning
        *  spiral. Note that the first evolved programs also made left-turning
        *  spirals. Coincidence? Actually, yes, which is why I was super
        *  confused at first, knowing that I commented this stuff out. Anyway,
        *  it's preserved here for nostalgia.
        */
        /*pos++;
        if (pos==length){ //if we've gotten as far as we wanted to
              turnCount = (turnCount + 1)%2;
              length+=turnCount;
              pos = 0;
              turnLeft();
        }else{// just keep going
        }*/

        theProgram.execute_void(0, new Object[0]);
        lastDirection = getDirection();

}

  /**
  * reset any local variables or anything here if you need to. It's called
  * when someone crashes and the player positions are reset.
  */
  synchronized void reset(){
        super.reset();
        length = 1;
        turnCount = 0;
        pos = 0;
  }

  /**
  * Just point in the current direction of movement. For example, if we were
  * going UP, look UP.
  * This doesn't produce a lot of great results. Maybe I'll try reintroducing
  * it at some point and see what happens.
```

```
*/
public void goStraight(){
      setDirection(lastDirection);
}

/**
* Aims the player in the direction that will get it to the opponent fastest
* (based on simple logic and elementary triangulation).
*/
public void moveTowardsOpponent(){
      //get closest enemy location
      int pIndex = 0;
      while (field.players[pIndex]==this){
            pIndex++;
      }

      Point p = field.players[pIndex].getLocation();

      int enemyX = p.x;
      int enemyY = p.y;

      //predict their next move
      int enemyDirection = field.players[pIndex].getDirection();

      //I hate switch statements, but this system is filled with 'em
      switch (enemyDirection){
            case Field.UP:
                  enemyY--;
                  break;
            case Field.DOWN:
                  enemyY++;
                  break;
            case Field.RIGHT:
                  enemyX++;
                  break;
            case Field.LEFT:
                  enemyX--;
                  break;
      }

      /*
      for each direction, calculate the distance between you in the next
      round. Use the smallest value. Note this doesn't take walls into
      account, so it may very well put you in your own tail.
      */
      Point me = getLocation();

      int t, t1;
      double t2;
      double nearest;
```

```
      t = Math.abs( me.x - enemyX ); //distance on X axis
      t1 = Math.abs( (me.y-1) - enemyY ); //distance on Y axis if we move UP
      nearest = Math.sqrt(t*t + t1*t1); //triangulated distance between us
      setDirection(Field.UP); //we'll assume that was closest,
                                        //then check the other dirs

      t1 = Math.abs( (me.y+1) - enemyY ); //if we move DOWN
      t2 = Math.sqrt(t*t + t1*t1);
      if (t2 < nearest) {
            nearest = t2;
            setDirection(Field.DOWN);
      }

      t1 = Math.abs( me.y - enemyY ); //reset Y var so we can check X

      t = Math.abs( (me.x-1) - enemyX ); //if we move LEFT
      t2 = Math.sqrt(t*t + t1*t1);
      if (t2 < nearest) {
            nearest = t2;
            setDirection(Field.LEFT);
      }

      t = Math.abs( (me.x+1) - enemyX ); //if we move RIGHT
      t2 = Math.sqrt(t*t + t1*t1);
      if (t2 < nearest) {
            nearest = t2;
            setDirection(Field.RIGHT);
      }
}

/**
* Aims you in the same direction as your opponent. This is actually kind of
* incorrect logic... if the opponent is moving towards you, "following" it
* will actually mean "leading" it. I'm as yet undecided on whether to aim
* for the opponent in that case. I probably should.
*/
public void followOpponent(){
      int pIndex = 0;
      while (field.players[pIndex]==this){
            pIndex++;
      }
      setDirection(field.players[pIndex].getDirection());
}

/**
* Just pick any old direction and move in it, provided there's no wall
* there. The inclusion of this in the genetic process has generally given
* unfavorable results. In many cases, evolving many generations will
* just give (move RANDOM) as the best fit candidate. Which is ridiculous,
* of course. I had hoped it would introduce some unpredictability, but it
* hasn't worked out like I planned. It kind of showcases the weakness of
```

```
  * the other commands and connections available (and maybe the fitness
  * measurement?) if RANDOM is the best you can come up with.
  *
  * Note: I had to override randomMove() in Player, due to the fact it
  * doesn't check to see if there are any possible moves first, resulting
  * in an infinite loop when the player is trapped. It doesn't cause
  * problems when you're in a threaded environment, but when everybody is
  * just waiting for you to die...
  */
protected int randomMove() {
    int dir = -1;
    boolean free = false;

        int x, y;
        Point p = getLocation();
        x=p.x; y=p.y;

        if (!field.isFree(x, y - 1) &&
              !field.isFree(x, y + 1) &&
        !field.isFree(x - 1, y) &&
        !field.isFree(x + 1, y)
        ) return -1;

        Random random = new Random(System.currentTimeMillis());

    while (!free) {
        dir = random.nextInt() % 4;

        switch (dir) {
        case Field.UP:
            free = field.isFree(x, y - 1);
            break;
        case Field.DOWN:
            free = field.isFree(x, y + 1);
            break;
        case Field.LEFT:
            free = field.isFree(x - 1, y);
            break;
        case Field.RIGHT:
            free = field.isFree(x + 1, y);
            break;
        }
    }

    return dir;
}

  /**
  * Just like moveTowards, but opposite. See comments therein.
  */
  public void moveAwayFromOpponent(){
```

```
//get closest enemy location
int pIndex = 0;
while (field.players[pIndex]==this){
      pIndex++;
}

Point p = field.players[pIndex].getLocation();

int enemyX = p.x;
int enemyY = p.y;

int enemyDirection = field.players[pIndex].getDirection();

switch (enemyDirection){
      case Field.UP:
            enemyY--;
            break;
      case Field.DOWN:
            enemyY++;
            break;
      case Field.RIGHT:
            enemyX++;
            break;
      case Field.LEFT:
            enemyX--;
            break;
}

Point me = getLocation();

int t, t1;
double t2;
double farthest = -1;

t = Math.abs( me.x - enemyX ); //distance on X axis
t1 = Math.abs( (me.y-1) - enemyY ); //distance on Y axis if we move UP
t2 = Math.sqrt(t*t + t1*t1); //triangulated distance between us
if (t2 > farthest){
      setDirection(Field.UP);
      farthest = t2;
}

t1 = Math.abs( (me.y+1) - enemyY ); //if we move DOWN
t2 = Math.sqrt(t*t + t1*t1);
if (t2 > farthest) {
      farthest = t2;
      setDirection(Field.DOWN);
}

t1 = Math.abs( me.y - enemyY ); //reset Y var so we can check X
```

```
            t = Math.abs( (me.x-1) - enemyX ); //if we move LEFT
            t2 = Math.sqrt(t*t + t1*t1);
            if (t2 > farthest) {
                  farthest = t2;
                  setDirection(Field.LEFT);
            }

            t = Math.abs( (me.x+1) - enemyX ); //if we move RIGHT
            t2 = Math.sqrt(t*t + t1*t1);
            if (t2 > farthest) {
                  farthest = t2;
                  setDirection(Field.RIGHT);
            }
      }
}
```