CSE 231 Spring 2010

Honors Programming Project 11

This assignment is worth 70 points and must be **completed and turned in on 4/26/2010 before midnight**

Assignment Overview

This project will give you experience on design and use of your own classes.

Project Description / Specification

This project uses a genetic algorithm (GA) to evolve a robot controller that follows a breadcrumb trail. The code for the robot and robot world are provided for you in file world.py. You need to write two classes:

- A GA class that runs the genetic algorithm.
- An individual class that represents a robot controller.

The code we have been working on in lab should be modified and used as part of this project. Additionally, code that implements the world and how a GA uses a matrix to figure out how the robot is moving has been provided for you.

0. Take a look at the world class (provided in world.py). It takes as input a file that represents the robot's world. World1.txt is an example world. Within the world, 'dim' is the number of rows in the world. The number of columns is inferred. Within the world, 'x' represents a wall, ' b' represents a breadcrumb (part of the trail), '-' represents a space in the room., 'F' represent the finishing spot, 'A' represents the agent, and '=' represents a square previously visited by the agent.

   The robot should follow a breadcrumb trail. A robot has 2 wheels – one on its left side and one on it right. Using its wheels, the robot is able to move forward one space, left one space, or right one space.

   A robot has 3 sensors – one to its left, one ahead of it, and one to its right. You can also query if it is on a breadcrumb, a dash, or the finish mark.

   More on this later…

1. **Create the individual class.** Use the code we developed in lab as a starting point. In this case, an individual is going to represent a robot controller as a Braitenberg matrix, which will transform sensor inputs to outputs

representing how the wheels should move. The matrix has 2 rows (one for each wheel) and 3 columns (one for each sensor).

At a high level, the robot decides how to move by multiplying its sensor values by the matrix (using matrix multiplication) and producing two outputs: (0) how the left wheel should move and (1) how the right wheel should move. This part of the algorithm has been implemented for you.

This class should have the following methods:

a. **__init__ (self)**
   Create a Braitenberg matrix as **a list of 6 numbers**. Initialize each value in the matrix to be between -5 and 5. (This part is similar to Honors Lab Part 1, except for that lab, the individuals were bitstrings.).
b. **mutate (self, mutation_rate)**
   This method should mutate an individual. Specifically, with the given mutation rate probability, it should change a value in the matrix to a different number between -5 and 5. It should produce the new mutated individual as output. (This part of the project is described in Honors Lab 3.)
c. **crossover (self, other)**
   This method should take in two individuals and using two-point crossover produce two offspring individuals as output. (This part of the project is described in Honors Lab 3.)
d. **evalFitness (self)**
   This method is the most complex one. To test how a matrix works at controlling the robot, we need to see how many new squares ('-') and breadcrumbs ('b') the robot visits in 25 moves.

   Each move:
   - Get the sensor values from the world
   - Determine what the robot should do. To help with this part, a function computing how the controller wants to move its wheels based on the present Braitenberg matrix and senor inputs is provided for you in test.py. It is called **getActuators.**

     getActuators produces 2 values by multiplying the sensor inputs by the matrix. These 2 values represent what the left wheel does and what the right wheel does.
     - If the values for both the left and the right wheels are positive, the robot moves 1 square forward.

    - If the values for the **left but not** the right wheel are positive, the robot moves 1 square to the right.
    - If the values for the right but not the left wheel are positive, the robot moves 1 square to the left.
- Move the agent in the world
- Check the result.
    - If the agent is on a breadcrumb it gets 5 points.
    - If the agent is on a dash it gets 1 point
    - If the agent is on the finish it gets 10 points. The agent may continue to move about the world after reaching the finish.

  (You can see the progress of the agent by printing the world.)

2. **Create the GA class.** Use the code we developed in lab as a starting point and refer to the Wikipedia article to better understand GAs. Your GA should have: This class should have the following methods:

    a. **__init__ (self, popSize, offspringPopSize, mutOffspring, mutBit),** where **popSize** is the population size, **offspringPopSize** is the number of offspring to create, **mutOffspring** is the percent of offspring to mutate, and **mutBit** is the probability a bit is mutated.

    In this method, initialize a population of size 200.

    tournamentSelection(self, n, k) (This part of the project is described in Honors Lab 4.)

    b. **run(self)**
    We are going to run the GA for **20** generations. For each generation, we need to:
        - Create offspring. Create an empty list of individuals. Until you have **offspringPopSize** individuals, select two parents using **tournamentSelection(2,1)**, cross them over, and add them to the offspring population.
        - Mutate some of the offspring. Mutate **mutOffspring** percent of the offspring.
        - Add the offspring to their parents' population.
        - Create the next generation. Using **tournamentSelection**(2,1) select individuals and add them to the next generation population.
        - Overwrite the original population with the next generation population.
        - Print the best fitness of this population to a file.

    Once the run is over, print the world of the best individual.

c. **EXTRA CREDIT: Implement this second selection method and use it, rather than tournament selection to select the parents.**

fitnessProportionateSelection(self)
For this method, we will select organisms proportional to their fitness. This function is sometimes referred to as roulette wheel selection. To perform fitnessProportionateSelection,

- sum the fitness values of all the individuals in the population.
- Pick a randomValue between 0 and the total fitness.
- Initialize a sumFitness value to 0
- For each individual:
    - sumFitness += fitness of individual
    - if sumFitness <= randomValue:
        - Save this individual as selected
        - Break
- Return the selected individual

3. **Run your GA** with the following parameters:
    a. GA(200, 50, 50, 25) – these parameters should enable you to evolve a working robot. Recall that this translates into:
    __init__ (self, popSize=200, offspringPopSize=50, mutOffspring=50, mutBit=25),

HINTS:

1. Start early.
2. Make sure each method works in isolation.
3. Work on the GA part first. Check that each part works by running it on the all ones problem that we discussed in class prior to moving on to the robot problem.