Modern Robots: Evolutionary Robotics

Programming Assignment 7 of 8*

Description

In this assignment you will evolve the morphology of block based robots, whose morphology is determined by a Compositional Pattern Producing Network (CPPN), to make them move as far as possible.

Tasks

1. Copy all your code for assignment 6 to the folder you will use for assignment 7. Make sure to keep a working backup of assignment 6.

2. Extract `hw_7_code`, and move all files to your assignment 7 folder.

3. First, you'll need to implement a CPPN. To implement a CPPN you can extend your neural network implementation, but you'll have to make the following changes: the network has to be *feed-forward* (meaning it can not contain any cycles, and each neuron needs to have an evolvable activation function. For this assignment, you will create a new neuron class where each neuron can have one of the following activation functions: linear, sine, Gaussian, and sigmoid. Open `Ind_CPPN_Neuron.hpp` and implement the `propagate` function, such that it will use one of these functions based one the value of the `_activationFunction` attribute.

4. Next, you have to make sure that the activation function can be evolved, and that the network is always feed-forward. Open `Ind_CPPN.hpp` and implement the following functions:

- `CPPN` (constructor). The `CPPN` constructor needs to set the number of inputs and outputs of the network, define a default value for the `_neuronActMutRate` (0.05 is suggested for this assignment), and add connections to the network such that all inputs connect to all outputs (and remember that the network has to be feed-forward, so backwards connections are not allowed).
- `randomize`. The `randomize` function should perform the same operations as the one in your neural network implementation, but also has to select a random activation function for each neuron.
- `mutate`. The `mutate` function also needs to perform the same operations as the one in your neural network implementation, and each neuron should have a `_neuronActMutRate` chance to randomly get a new activation function. In addition, whenever you add a new connection to the network, you need to make sure that this action does not cause a cycle.

*Original material was graciously provided by Josh Bongard. Jeff Clune slightly modified it. Joost Huizinga heavily modified it.

- `getDepth`. The `getDepth` function should return the depth of the network, which is the number of steps required to activate the entire network.

In all cases, see the comments in the header file for details. Once you have implemented the CPPN you can use the `checkCPPN` function found in `Test_HW7UnitTest.hpp` to test that your CPPN implementation works correctly.

5. Next you'll need to create an evolving robot. Open `Ind_EvolvingRobot.hpp`, and implement the following methods:

- `build`. This function should build the robot based on its CPPN.
- `_connect`. This is a helper function called by build above; it should connect two adjacent blocks for our robot.
- `_activateCppn`. This is a helper function which will activate the CPPN.

Detailed instructions can be found in the comments of each function. This should allow you to successfully build a robot given a CPPN genome.

6. You'll also need a slightly different fitness function to evolve these robots. Implement `Fit_EvolvedRobot.hpp` according to the comments. You can copy most of your code from previous assignments, but you'll have to make some minor changes to account for the fact that an individual is a robot, not a neural network.

7. Now it is time to evolve the robots. Open `main.cpp`, comment out or delete all previous code (backup!) and include all necessary header files.

Your individual should have the type:

```
typedef IndivFitness<EvolvingRobot> ind_t;
```

Try the following parameters:

- Population size: 50
- Generations: 10
- Parent selection: tournament selection with tournament size 10
- Survivor selection: tournament selection with tournament size 10

Log the fitness over time and write it to a file. Plot this file with `plotLine.py`. Also make a screen shot of the final individual after it moved at least some distance. Your figures should look like figures 1 and 2.

Deliverables

A pdf document containing the figures resembling figures 1 and 2 and any files you changed.

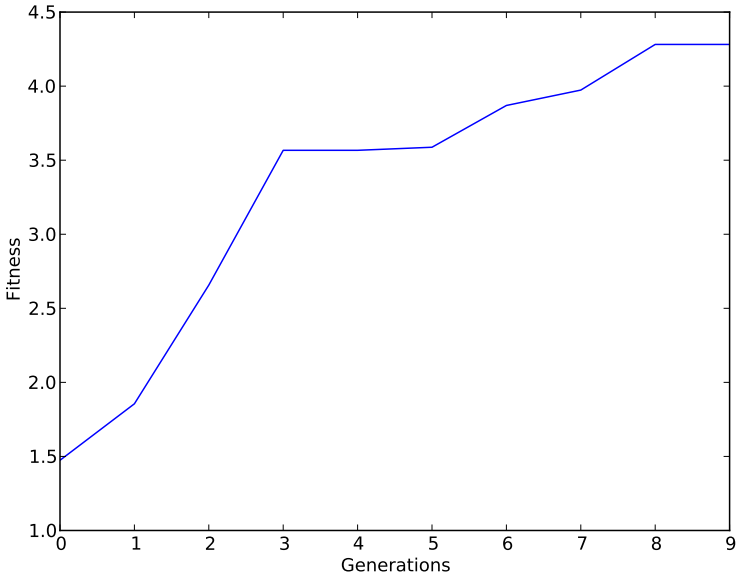

Figure 1: The fitness of the best individual over time. Note: while your fitness curve can be quite different, your algorithm should at least be able to reach a fitness of 4. If it does not, try re-running your algorithm a few times. If it doesn't reach a fitness higher than 4 in any of those runs, there is probably a bug in your code.

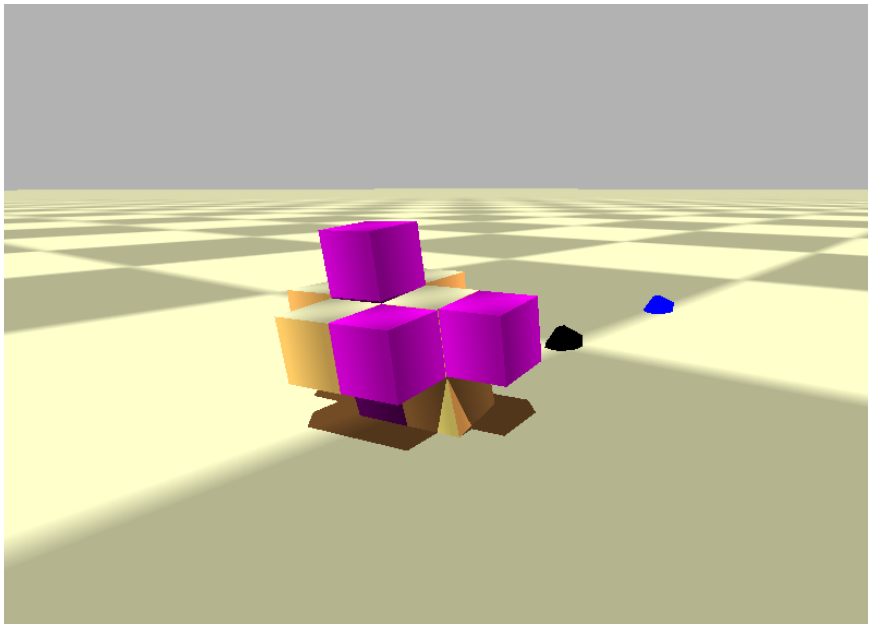

Figure 2: The final individual. Your individual may look quite different, but do make sure that it moved at least some distance from the starting location (the black dot).