# CSG2341 Intelligent Systems

## Workshop 9: Genetic programming – evolving a program for a robot

### Related Objectives from the Unit outline:

- Identify appropriate intelligent system solutions for a range of computational intelligence tasks.
- Demonstrate the ability to apply computational intelligence techniques to a range of tasks normally considered to require computational intelligence.

### Learning Outcomes:

After completing this workshop, you should be able to demonstrate an understanding of the resources required to implement a problem solution based on genetic programming, to describe and analyse the steps in designing a genetic programming algorithm to solve an optimisation problem (in this case, to design a control program for a robot) and use a computer package to implement a genetic programming algorithm to design such a program.

---

### Background

In the text book and lectures, you have seen genetic programming being used to evolve an approximation to a function. In this workshop, you will see a different example, evolving a program for a "robot".

The robot is placed in a rectangular field, and its task is to visit as much of the field as possible in a set amount of time. Figure 1 shows the robot in its field. The white areas are free space, while the grey shows the boundary of the field and a number of obstacles inside the field. The robot is the blue disk, and the red dot inside it shows which direction the robot is facing. The black path shows where the robot has been. The green lines show the robot's vision system scanning in each direction.

At each time step, the robot can do one of four things

- move forward a step;
- move back a step;
- turn left; or
- turn right.

If the robot tries to move forward or back, but is blocked by an obstacle or the boundary, it stays put. When the robot visits a position, it leaves a detectable trace.

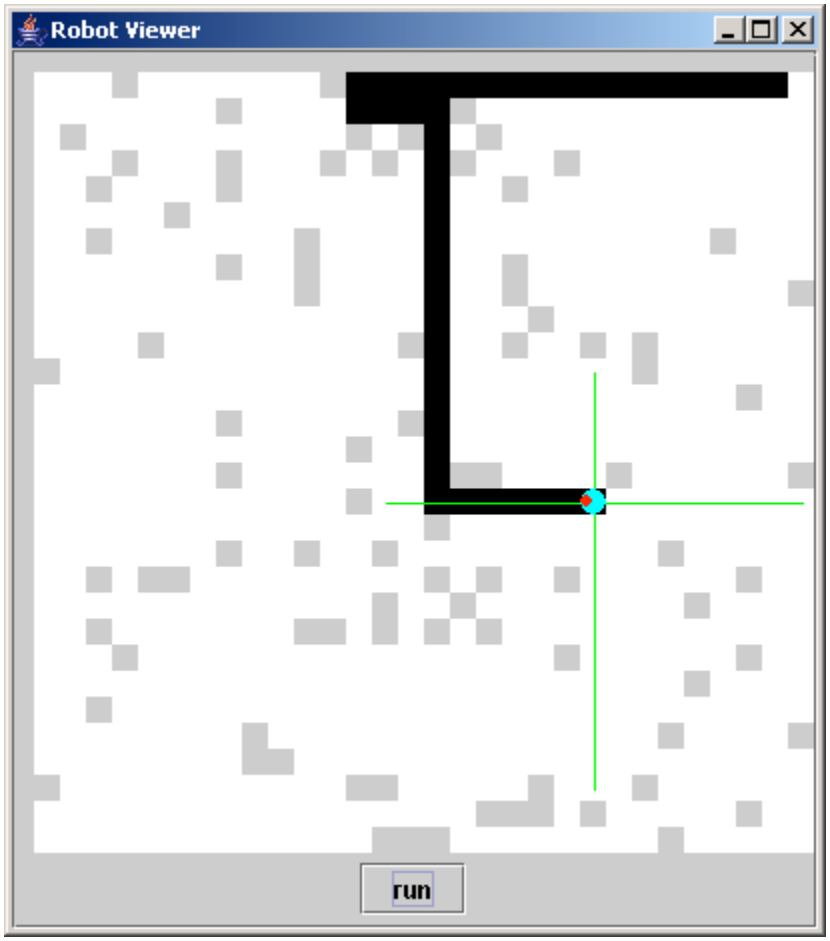



**Figure 1 - The robot in its field**

The robot also has four directional detectors, which tell it how many unvisited squares there are between it and an obstacle or the boundary, in each of the four directions. The direction that the robot is facing in called “north”, behind it is “south”, to its left is “west” and to its right is “east”. The four directional detectors can also tell the robot whether each adjacent square is empty and not visited.

The robot can also decide between two actions with a certain probability.

The robot is controlled by a program that tells it what action to take at each time step. The program can use the following programming constructs:

*Basic actions:*

- `forward` – step forward
- `backwards` – set backwards
- `left` – turn left
- `right` – turn right

*Directions*

- `north` – the direction the robot is facing
- `south` - behind the robot
- `east` – to the right of the robot
- `west` – to the left of the robot

*Integers*

- `0` to `size-1`, where the field is `size` wide and `size` high

*Doubles*

- `0` to `1.0`

*Booleans*

- `true` or `false`

*Functions*

- `isGood(dir)` – whether the next square in direction `dir` is empty and has not been visited.
- `free(dir)` – the number squares not yet visited before the boundary or an obstacle in direction `dir`
- `chance(prob)` – `true` with probability `prob`, otherwise `false`
- `if(condition, action1, action2)` – if `condition` is `true`, do `action1`, else do `action2`
- `and(condition1, condition2)` – logical and of the two conditions
- `or(condition1, condition2)` – logical or of the two conditions
- `not(condition)` – logical negation of `condition`
- `equals(int1, int2)` – `true` if `int1` == `int2`
- `<` – `true` if `int1` < `int2`
- `>` – `true` if `int1` > `int2`

The program takes the form of a tree using these constructs. Figure 2 shows a genetic programming algorithm running, and the best program tree so far found by the algorithm.

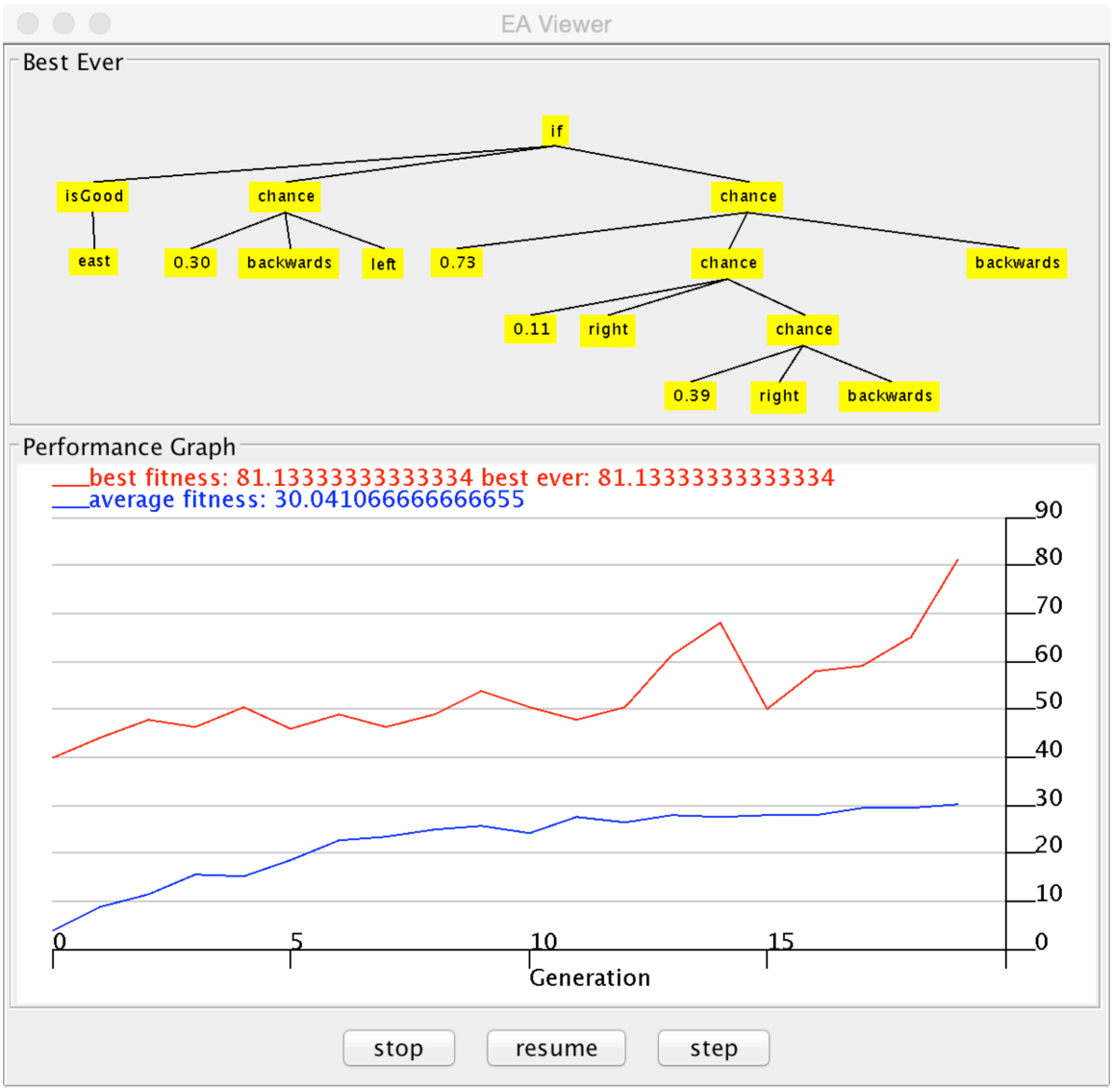



**Figure 2 - GP algorithm running, showing the current best program**

A robot running the program in the figure would do the following:

1. The topmost node of the program tree is an “IF”. So the robot must evaluate the first branch, which will return either “true” or “false”. If “true”, then the robot would execute the second branch of the tree – if “false” it would execute the third branch.

   In this case, the left branch (isGood east) checks to see if the square to the robot’s right is free and has not been visited.

   a. If so then it follows the second branch of the tree (chance 0.3 backwards left). So 30% of the time, the robot would go backwards, and 70% of the time would go left.

b. If the square to the robot's right is not free or has been visited, then the third branch of the tree will be followed, i.e. (chance 0.73 (chance 0.11 right (chance 0.39 right backwards)) backwards)
    i. So 73% of the time, (chance 0.11 right (chance 0.39 right backwards)) is followed, and
    ii. 37% of the time, the robot goes backwards.

In this way, starting at the top of the tree, and doing calculations and tests to see which branch to follow, eventually, the robot reaches a leaf node : in this case a left, a right, or backwards, and executes that action.

On average, a robot following this program would cover just over 81 squares of the maze in the allowed number of time steps – and this is its "fitness".

**Task:**

Your task is to investigate the ability of a genetic programming algorithm to find good programs for the robot.

**Step 1**

Download EvolveRobot.zip from BlackBoard, unzip into a working directory, and load into a NetBeans project.

**Step 2**

Build the project and run EvolveRobot.

This will open two windows, like those in Figure 1 and Figure 2. The first window is the usual EvolutionaryAlgorithm window, and the second one will show the best evolved robot (when the genetic programming algorithm finishes running). You might have to look behind the first window to find it. You can run the robot again by clicking on "run".

**Step 3**

The java source file EvolveRobot.java contains the following code segment:

```
private static final int SIZE = 30;     // the field is size x size
private static final int OBSTACLES = 300;// number of obstacles
                                         // on the field
private static final int NSTEPS = 450;  // number of actions the
                                         // the robot is allowed
```

By changing these and recompiling, you can change the environment in which the robot operates.

Run the genetic programming algorithm a few times, say 10, with each of the following settings, recording the best control program found for each setting:

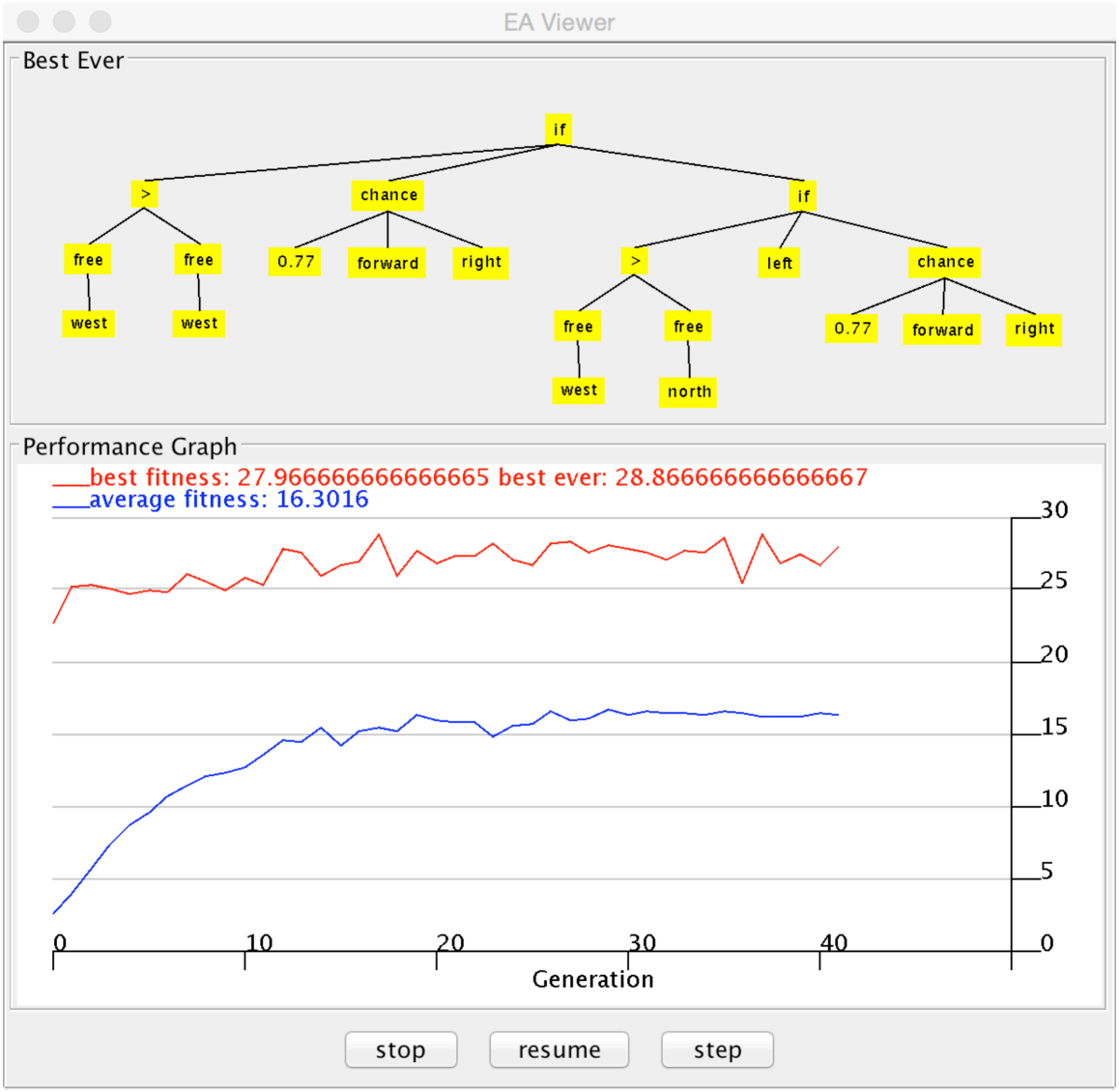
```
Case 1.  SIZE = 10, OBSTACLES = 10, NSTEPS = 100

Case 2.  SIZE = 10, OBSTACLES = 0, NSTEPS = 100

Case 3.  SIZE = 10, OBSTACLES = 25, NSTEPS = 100
```

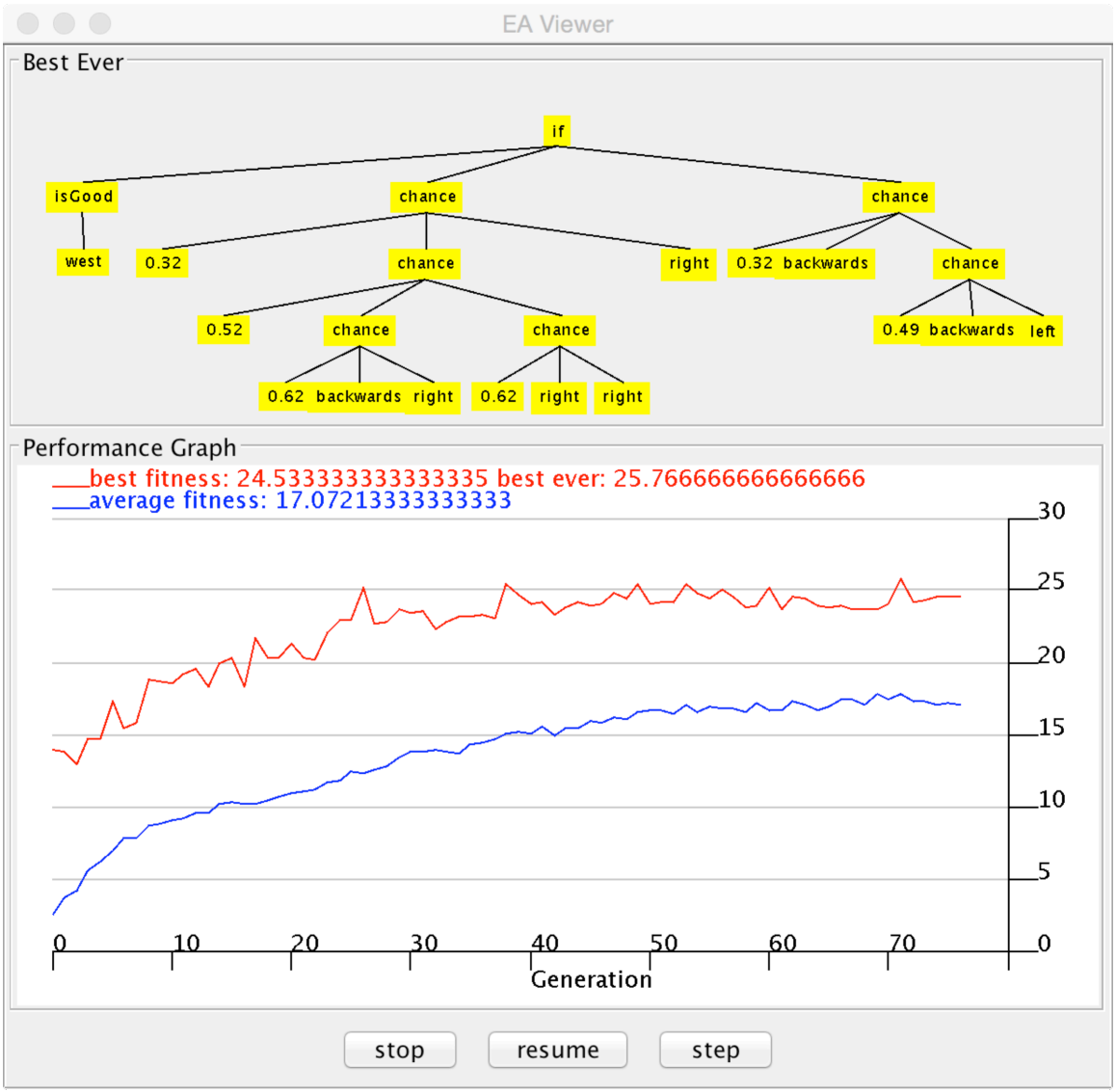
Now answer the following questions:

***Question 1:*** Many of the evolved programs will be larger than they need to be. For example:




in this tree, the first branch test (< (free west) (free west)) is always false (why?) so only the third branch will be used, so the whole tree could be replaced with that branch.

Or in this one :




The subtree (chance 0.62 right right) will always go right, so can be replaced with (right).

Other simplifications are possible – can you think of any? This problem of extraneous code in evolved programs is called “bloat”. It slows down the genetic programming algorithm.

For each of the best control programs you found above, remove all unnecessary parts of the tree. Now try to describe in your own words what each program is doing.

***Question 2:*** Describe any differences that you observe in the best programs for cases 1, 2 and 3 and suggest an explanation of the differences. (Of course, a lot more experimentation would be needed to confirm your observations, and further reasoning or other experiments would be needed to support your explanation.)

*Note:* the fitness values will be different for the different cases – this does not mean anything – the tasks are different so you cannot compare fitness values.

***Question 3:*** Suggest a way to improve the genetic programming algorithm by reducing the effect of the bloat problem.

**When you have completed this workshop, submit your answers to the three questions to your tutor using the submission facility on BlackBoard.**