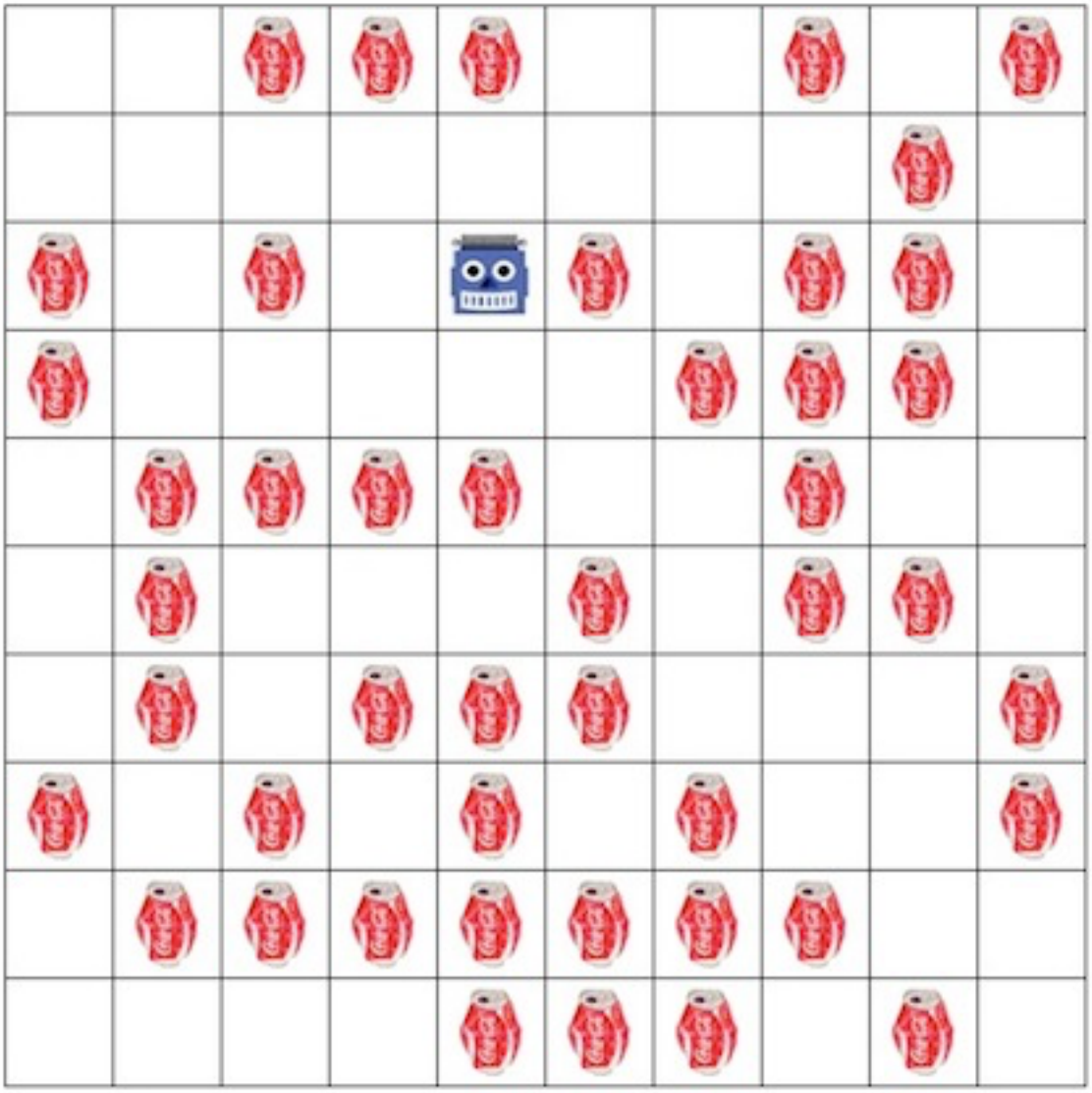
# A Genetic Algorithm for Robby the Robot

*Project created by James Marshall, Computer Science, Sarah Lawrence College.*



In this assignment, you will implement a genetic algorithm to evolve control strategies for Robby the Robot, as described in Chapter 9 of *Complexity: A Guided Tour* by Melanie Mitchell.

***Note:*** **The simulator code for this project only works with Python 2. You can use the usual tools to manage your Python versions.**

A control strategy will be represented as a string of characters that code for the following robot actions:

- **0** = MoveNorth
- **1** = MoveSouth
- **2** = MoveEast
- **3** = MoveWest
- **4** = StayPut
- **5** = PickUpCan
- **6** = MoveRandom

Your GA will maintain a population of genomes representing control strategies. Each genome will be a 243-character string specifying the action that Robby should take for every possible situation he might find himself in. Robby's "situation" will be encoded as a 5-character *percept string* specifying what Robby currently sees in his immediate vicinity. For example, the string `'WECWC'` means there is a wall to the north, an empty grid cell to the south, a soda can to the east, another wall to the west, and a soda can in Robby's own grid cell. Each percept string corresponds to a unique code number in the range 0-242, which indicates the (0-based) index number of the situation, as illustrated in Table 9-1 on page 132 of the handout. The job of your GA is to evolve a good control strategy that maximizes Robby's average cumulative reward as he collects empty soda cans for 200 time steps.

**Robby's World**

To use the simulator, download the file **robby.zip** and unzip it. This will create a folder named **robby**. Put this folder in the same location as your Python program for this assignment.

*IMPORTANT: do not put your Python file or any other files inside the folder; just make sure the folder is in the same location as your program file. You should not modify any of the code in this folder.* Then put the following lines at the top of your program file, which will create a 10 × 10 grid world named `rw` when your file is loaded into Python:

```
import robby
rw = robby.World(10, 10)
```

To interact with the world, you can use the following commands:

- **rw.getCurrentPosition()** returns the current 0-based *row*, *column* position of Robby in the grid.
- **rw.getPercept()** returns the percept string specifying what Robby currently sees to the North, South, East, West, and Here.
- **rw.getPerceptCode()** returns the code number of the current percept string as an integer in the range 0-242.
- **rw.distributeCans(*density*=0.50)** randomly distributes soda cans throughout the world, with $0 \leq density \leq 1$ specifying the probability of a can occupying a grid cell. The default value is 0.50.
- **rw.goto(*row*, *column*)** moves Robby to the specified grid location.

- **rw.performAction(*action*)** causes Robby to perform the specified action, where *action* is one of the strings "MoveNorth", "MoveSouth", "MoveEast", "MoveWest", "StayPut", "PickUpCan", or "MoveRandom".

The following abbreviations are provided for convenience:

| | | | |
|---|---|---|---|
| **rw.north()** | = rw.performAction("MoveNorth") | **rw.stay()** | = rw.performAction("StayPut") |
| **rw.south()** | = rw.performAction("MoveSouth") | **rw.grab()** | = rw.performAction("PickUpCan") |
| **rw.east()** | = rw.performAction("MoveEast") | **rw.random()** | = rw.performAction("MoveRandom") |
| **rw.west()** | = rw.performAction("MoveWest") | **rw.look()** | = rw.getPercept() |

- **rw.graphicsOff(*message*="")** turns off the graphics. This is useful for simulating many actions at high speed when evaluating the fitness of a strategy. The optional *message* string will be displayed while the graphics are turned off.
- **rw.graphicsOn()** turns the graphics back on, and updates the grid to reflect the current state of the world.
- **rw.show()** prints out a non-graphical representation of the current state of the world.
- **rw.demo(*strategy*, *steps*=200, *init*=0.50)** turns on the graphics and demos a strategy for the specified number of simulation steps (default 200) with a randomized soda can density of *init* (default 0.50). Optionally, *init* can be a string specifying the name of a grid world configuration file created with the save command.
- **rw.save(*filename*)** saves the current grid world configuration as a text file, where *filename* is a string.
- **rw.load(*filename*)** loads a grid world configuration from a text file, where *filename* is a string.
- **rw.strategyM** is a string representing the hand-coded strategy *M* created by Melanie Mitchell, as described in the handout.

For example, you can watch strategy M in action by typing the command `rw.demo(rw.strategyM)` at the Python prompt.

### GA Implementation Details

I would recommend using the following starting parameters for your GA:

- Population size: 200
- Crossover rate: 1.0 (meaning 100% probability of single-point crossover)
- Mutation rate: 0.005 per locus
- 200 actions per cleaning session
- Number of generations: 500

The fitness value of a control strategy should be the average total reward accumulated during a cleaning session when following the strategy, averaged over 100 sessions.

For this assignment, you should use *rank selection* when selecting genomes for crossover and mutation. Many control strategies can have negative fitness values, which would cause problems with fitness-proportionate selection.

To implement rank selection, first calculate the fitness of each genome in the population. Then sort the genomes by their fitness values (whether positive or negative), in increasing order from lowest to highest fitness. The selection weight of a genome will be its rank number from 1 (lowest) to 200 (highest), instead of the fitness value itself. Here is one way in Python to sort a list of genomes based on fitness. Assuming that a function called `fitness` is defined for genomes, the function `sortByFitness` shown below takes a list of genomes (strategy strings) and returns (1) a new list of the genomes sorted into ascending order by fitness value, and (2) a list of the corresponding fitness values:

```
def sortByFitness(genomes):
    tuples = [(fitness(g), g) for g in genomes]
    tuples.sort()
    sortedFitnessValues = [f for (f, g) in tuples]
    sortedGenomes = [g for (f, g) in tuples]
    return sortedGenomes, sortedFitnessValues
```

**Experiments**

Run your GA for a total of 500 generations. Every 10 generations, your GA should record the best strategy from the current population, along with the strategy's fitness value, in an output file called **GAoutput.txt**. More specifically, each line of the file should contain four items separated by whitespace, in the following order: (1) the generation number, (2) the average fitness of the whole population for this generation, (3) the fitness value of the best strategy of this generation, and (4) the best strategy itself. You may also want to have your GA periodically demo the best strategy found so far (every 10 or 20 generations, for example). That way, as the evolution proceeds, you can watch Robby's progress as he learns to clean up his environment.

You should also experiment with different GA settings (population size, number of generations, crossover and mutation rates, etc.) to see how quickly your GA can discover a really good strategy. You can save your best strategies to text files.