Simulating Language: Lab 4 Worksheet

This simulation implements the ***evolution*** of an innate signalling system, using the same basic signalling system code from last time. Make sure that you are familiar with the way in which agents and signalling systems were encoded; the same data structures are used here. On this worksheet, the program is relatively long, so only the **new code** is reproduced. The file `evolution1.py` nevertheless contains all the code we have already seen in `signalling2.py`. Copy `evolution1.py` from the website, and save it to your own file space as before.

Copying Lists

```
from copy import deepcopy
```

The first part of the new code imports the **deepcopy** function; this is needed because of the way in which Python treats copies of lists. Have a look at the code in the example below, and see if you can understand what is happening.

First, list `a` is created, then is 'copied' to `b`, then one of the values in `a` is changed. But note that the value in `b` is also changed!

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[1] = 5
>>> b
[1, 5, 3]
```

When copying compound objects (i.e. lists), by default Python fills the new list (here: `b`) with references to elements in the old list (`a`); this means that the contents of `b` is actually **the same** as that of `a`, even if we change `a` after we 'copied' it.

```
>>> from copy import deepcopy
>>> x = [1, 2, 3]
>>> y = deepcopy(x)
>>> y
[1, 2, 3]
>>> x[1] = 5
>>> y
[1, 2, 3]
>>> x
[1, 5, 3]
```

If, instead, we want to ensure that the copied list contains new and different items, then we need to make a **deep copy**, using the deepcopy function from the copy module rather than simple assignment. Look at the next example to see how this works.

Make sure that you understand the difference, given a list `x`, between the statements `y = x` and `y = deepcopy(x)`.

Simulation Parameters

The next section defines a number of variables which are used as parameters in the simulation, with comments explaining what they are used for (remember that anything after the hash sign (#) is a comment, and thus ignored by the Python interpreter). We define the variables individually, and then refer to them by name in the following functions, so that when we want to run the simulation

```
mutation_rate = 0.01   # probability of mutation per weight
mutation_max = 1       # maximum value of a random weight
send_weighting = 10    # weighting factor for send score
receive_weighting = 10 # weighting factor for receive score
meanings = 3           # number of meanings
signals = 3            # number of signals
interactions = 1000    # number of interactions per generation
size = 100             # size of population
```

with different parameters, all we need do is either change the values here and re-run the module, or enter new values at the prompt in IDLE and run a new simulation.

How would you change the number of agents in the population?

Fitness Functions

Evolutionary algorithms require a function which measures fitness and helps determine which agents will reproduce into the next generation. The following functions define fitness for an individual agent (`fitness`) and for the whole population (`sum_fitness`); study them and make sure you understand how they work.

```
def fitness(agent):
    send_success = agent[2][0]
    send_n = agent[2][1]
    receive_success = agent[2][2]
    receive_n = agent[2][3]
    if send_n == 0:
        send_n = 1
    if receive_n == 0:
        receive_n = 1
    return ((send_success/send_n) * send_weighting +
            (receive_success/receive_n) * receive_weighting) + 1

def sum_fitness(population):
    total = 0
    for agent in population:
        total += fitness(agent)
    return total
```

Why are the variables `send_n` and `receive_n` set to 1 in the `fitness` function?

What do the `send_weighting` and `receive_weighting` variables do?

What variables does the `fitness` function depend on? Why is there a "+1" here?

Mutation

This function mutates the signalling system by going through each cell in the matrix, deciding whether a mutation should take place, and, if so, assigning a new value to the cell. Note that this function contains a new random function **random.randint(x, y)**; this returns a random integer between x and y, **including both x and y**; `random.randint(x, y)` is therefore equivalent to `random.randrange(x, y + 1)`

```
def mutate(system):
    for row_i in range(len(system)):
        for column_i in range(len(system[0])):
            if random.random() < mutation_rate:
                system[row_i][column_i] = random.randint(0, mutation_max)
```

How does the program make sure that it goes through each cell in the matrix?

How frequently does mutation happen?

Breeding the next generation of agents

The following functions create a new population of agents based on the fitness of the existing agents. The probability of being picked as a parent agent is proportional to the agent's fitness. There is another new random function **random.uniform(x, y)**, which returns a random floating-point number between x and y; `random.uniform(0,1)` is equivalent to `random.random()`. Make sure you understand how the pick_parent function works.

```
def pick_parent(population,sum_f):
    accumulator = 0
    r = random.uniform(0, sum_f)
    for agent in population:
        accumulator += fitness(agent)
        if r < accumulator:
            return agent

def new_population(population):
    new_p = []
    sum_f = sum_fitness(population)
    #print(sum_f) #uncomment this line if you would like updates during runs
    for i in range(len(population)):
        parent=pick_parent(population, sum_f)
        child_production_system = deepcopy(parent[0])
        child_reception_system = deepcopy(parent[1])
        mutate(child_production_system)
        mutate(child_reception_system)
        child=[child_production_system,
               child_reception_system,
               [0., 0., 0., 0.]]
        new_p.append(child)
    return new_p
```

How does the program ensure that the probability of being picked as a parent is proportional to fitness?

*Why is **`deepcopy`** used in `new_population`?*

Establishing a random population of agents

The function `random_system` generates a random signalling system, and this is used to generate a random population of agents (`random_population`).

```
def random_system(rows,columns):
    system = []
    for i in range(rows):
        row = []
        for j in range(columns):
            row.append(random.randint(0, mutation_max))
        system.append(row)
    return system

def random_population(size):
    population = []
    for i in range(size):
        population.append([random_system(meanings,signals),
                           random_system(signals,meanings),
                           [0., 0., 0., 0.]])
    return population
```

Running the simulation

```
def simulation(generations):
    accumulator=[]
    population = random_population(size)
    for i in range(generations):
        for j in range(interactions):
            pop_update(population)
        average_fitness=(sum_fitness(population)/size)
        accumulator.append(average_fitness)
        population = new_population(population)
    return [population,accumulator]
```

This function runs the main simulation. Make sure that you understand how it works, by studying the above functions again if necessary. After having run this module (remember that you choose **Run-Run Module** in the editor to load the program into Python), run the simulation by simply typing `simulation(n)` at the prompt, where n specifies the number of generations you want to simulate.

> *How often does the population communicate in each generation?*
>
> *At what point are agents assessed for fitness?*
>
> *Run the simulation for a few generations: what do values returned by* `simulation` *signify?*
>
> *Run it again, with different numbers of generations: how long does it take for a stable, successful communication system to emerge? (Note: 1000 generations takes about 40 seconds on my laptop, so be wary of starting very very long runs)*

Questions

1. Under what conditions does stable, successful communication evolve? (Note that it is a very good idea to run the simulation a few times, and plot the results).

2. Can you speed up evolution (or slow it down)? How? Is there a limit to how fast evolution can happen in the model?

3. In earlier worksheets we gave you the option of modelling production and reception using a single matrix of weights, or of modelling populations in a more structured way (e.g. where each individual communicated with their neighbours). What difference do you think these factors will make to the evolution of communication? Make the necessary adjustments to the code and find out.

4. In this model a parent's signalling system is transmitted directly to their offspring - this is our model of the genetic transmission of an innate signalling system. How else might a signalling system be transmitted from parent to offspring, and how might you model that process?