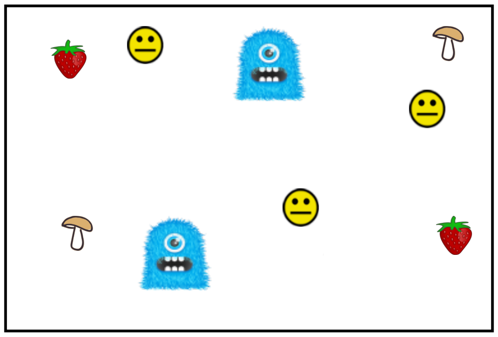
COSC343 Assignment 2

Evolve a species

Weight:14% Lecturer: Alistair Knott

For this assignment, you will implement a genetic algorithm to optimise the fitness of a species of **creatures** in a simulated two-dimensional world. The world should contain edible and poisonous **foods**, placed at random, and a population of **monsters** (your basic zombies). A whimsical illustration is given below. Your creatures are the smileys; the blue things are the monsters; the strawberries are edible food; the mushrooms are poisonous food. The algorithm should find behaviours that keep the creatures well fed and not dead.



Task 1: Implementation (5 marks)

You can implement the simulation in any language you like. The brief is very open: you can implement the above description however you like. I'll give a more detailed specification in this section, which you can implement to the letter or adapt as you wish.

Data representation

The locations of the strawberries and mushrooms are stored in two $n \times m$ arrays, `strawb_array` and `mushroom_array`, placed randomly (with the constraint that strawberries and mushrooms can't occupy the same square). Each cell of each array contains a number, indicating the quantity of food of the given type on the associated square.

Creatures are represented in a 1-dimensional array of `creature` structures. Each `creature` structure holds the **state** of the creature, which comprises an `energy_level`, and a `location` on the $n \times m$ grid. Monsters are represented in a similar 1-dimensional array of `monster` structures, each of which holds a `location`.

Creatures have senses that allow them to detect food and monsters in their local neighbourhood (see next section). Monsters can detect creatures in their local neighbourhood. To implement these functions, it's useful to store the locations of creatures and monsters in two additional $n \times m$ arrays: `creature_array` and `monster_array`. This makes it more efficient to find the items in any given neighbourhood: we just need to search the neighbourhood, rather than searching over all creatures/monsters. These arrays should hold integer values, representing the number of creatures/monsters on each square.[1]

Creature and monster capabilities

Each creature's `energy_level` is decremented by a small fixed increment at each timestep. If it runs out of energy it is **dead**.

A creature has some hardwired **senses**, telling it about things present in its current square, or in its current **neighbourhood**, at the current time step. (A neighbourhood is a square region of the world, centred on the creature's location.) The senses are implemented by a family of six sensory functions.

- `strawb_present` returns a Boolean (1 or 0) indicating the presence/absence of strawberries at the current location. `mushroom_present` does the same for mushrooms.
- `nearest_strawb` returns the direction to move in towards the nearest strawberry in its current neighbourhood. (Directions are North, East, South and West.) If there are no strawberries in the neighbourhood, the function returns zero. There are similar functions `nearest_mushroom`, `nearest_monster`, and `nearest_creature`.

A creature can also perform an **action** at each timestep. Each kind of action requires a certain amount of energy, and lowers the creature's `energy_level` by this amount.

- It can **move** (North, South, East, West, one square at a time). The direction of movement can be `random`, but it can also be relative to the directions returned by its senses: it can move `towards` or `away_from` the nearest X in its neighbourhood.
- It can **eat**—which will consume one unit of food from its current square (provided there is some). (To reflect this, the value of the relevant location in `strawb_array` or `mushroom_array` should be decremented by some constant after each eat action.) Eating a unit of strawberries increases the creature's energy by some fixed amount (greater than the energy required to do the 'eat' action). Eating a unit of mushrooms reduces its energy level to zero (i.e. kills it).

Monsters move towards the `nearest_creature` in their neighbourhood (or at random if there's no creature in the neighbourhod). So that monsters move slower than creatures, this operation only happens once every f timesteps. If a monster and a creature share the same frame, the creature dies.

[1]These values must obviously be kept in synch with the locations held in the `creature` and `monster` structures: when a creature moves from square $(10, 10)$ to $(10, 11)$, you must update the `location` in its `creature` structure, but also decrement `creature_array(10,10)` and increment `creature_array(10,11)`.

Creature chromosones

Each creature also has a **chomosone**, which specifies a mapping from the information gained through sensory functions to actions. There are 13 positions in the chromosone:

1	2	3	4	5	6	7	8	9	10	11	12	13

The first six positions specify the action associated with each of the sensory functions. The seventh position specifies a default action.

Posn	Role	Possible values
1	action to do when `strawb_present`	eat/ignore
2	action to do when `mushroom_present`	eat/ignore
3	action on `nearest_strawb`	towards/away_from/random/ignore
4	action on `nearest_mushroom`	towards/away_from/random/ignore
5	action on `nearest_monster`	towards/away_from/random/ignore
6	action on `nearest_creature`	towards/away_from/random/ignore
7	default action	random/north/east/south/west

Thus, position 1 specifies what the creature will do when it senses strawberries in its current square (either eat it or ignore it); position 3 specifies what the creature will do when it senses a nearby square with strawberries (move towards it, move away from it, or ignore it). Position 7 specifies the action to do if sensory stimuli aren't recommending any actions (i.e. if there are no sensory stimuli, or if the only stimuli present are ones for which the action is 'ignore'). Positions 8–13 hold numbers, that specify *weights* for actions 1–6, to determine what to do if multiple actions are activated.

8	action 1 weight	number
9	action 2 weight	number
10	action 3 weight	number
11	action 4 weight	number
12	action 5 weight	number
13	action 6 weight	number

For instance, if the chromosone sets up the creature to eat when it's in a location with strawberries (posn 1 = 'eat') and to move away from the nearest monster if there's one nearby (posn 5 = 'away from'), it needs a way of deciding what to do when it's at a strawberry *and* there's a nearby monster. This policy is implemented in the numbers at positions 8 and 12, that hold the weights of the alternative actions: if the number at position 8 is higher, it will eat, otherwise it will run away. (Note that if the action for a sensory condition is 'ignore', it's not in the competition.)

You will have to write code for your creature that implements the above specification. It will look something like this:

```
Procedure select_action

Initialise actions_list to the empty list

For sensory functions 1-6:
    Compute the result of the function
    If the result is non-zero:
        Calculate the associated action (by look-up in the chromosone)
        If the action is not 'ignore':
            Add the action to actions_list
If actions_list is empty:
    Do the default action (posn 7 in the chromosone)
Else:
    Determine which of the competing actions has the strongest weight
      (posns 8-13 in the chromosone)
    Return this action.
```

Creature evolution

You will simulate a number of **generations** of your species, each consisting of a **population** (of a fixed size p). Each simulation will last for a fixed number of time steps t. At each time step, each creature performs one action; monsters perform one action every f time steps.

At the end of each simulation, the **fitness** of each creature is determined by its `energy_level`. You will then create a new generation of creatures, of the same size as the previous generation (i.e. containing k individuals), by iteratively **selecting** two 'parent' creatures (based on their fitness) and creating a 'child' creature using **crossover** and **mutation** (as discussed in Lecture 12).

Implementation hints

- If you want to use Python for this assignment, you could use the animation method in the optimisation code from Tutorial 6. This lets you plot points and change their location or colour over time. (You could plot creatures as dots, with a colour dependent on their energy level, and plot food as dots with a colour or size dependent on quantity.) But feel free to use another animation method if you prefer.

- It's a good idea to get the behaviour of the creatures and monsters working before you start thinking about evolution, obviously.

Task 2: Report (5 marks)

You should also write a report about your simulation. The report should include:

- A description of your simulation. This just needs to specify how your simulation *differs* from the specifications set out here.
- A graph showing how average fitness of the population changes as evolution proceeds.
- A description of how evolution shaped your creatures' behaviour.

Marking scheme (Total: 10 marks)

Marks will be allocated as follows:

- Task 1: **5 marks**. This task will be assessed by a demo of your simulation during your tutorial in Week 11. In the demo, you must show me two things:
  - An animation of your world showing the behaviour of the initial population, before any evolution has happened;
  - An animation of the world showing the behaviour of the final population in your simulation, after several generations of evolution;

  (You can make movies of these if you like.) Marks won't be added for variations on the assignment spec, but they won't be subtracted either (provided your code implements the scenario in the very first paragraph).

  During the demo, you should be prepared to answer questions about the simulation.[2]
- Task 2: **5 marks**. Marks will be awarded for clarity of the report, and for addressing the topics you need to discuss.

Submission

The assignment is due at **4pm on Tuesday of Week 11** (**13 May**). You should submit your code for Task 1 electronically, using the `submit343` script, by 4pm that day. You should also submit your report for Task 2 on paper, to the assignments box near the office in the Owheo building, by the same deadline. You should include a *hard copy of the code (including comments)* as an appendix to this report.

The demo for Task 1 will be done in your tutorial in Owheo later that week.

For each task, you will lose 10% of available marks for each day late.

[2]Even if you don't get evolution working fully, marks will still be awarded for implementation of the appropriate creature and monster behaviours: i.e. even if you 'hard code' sensible behaviours, rather than learning them, you'll still get some marks.