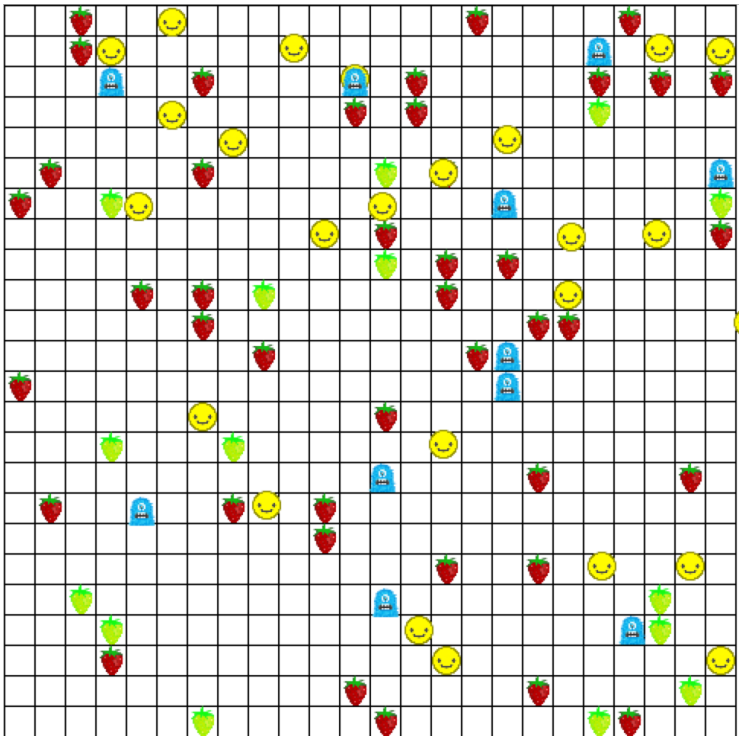
COSC343 Assignment 2

Evolve a species

Weight:20% Lecturer: Lech Szymanski

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 contains edible **foods**, placed at random, and a population of **monsters** (your basic zombies). An illustration is given below. Your creatures are the smileys; the blue things are the monsters; the strawberries are edible food (red for ripe and green for non-ripe). The algorithm should find behaviours that keep the creatures well fed and not dead.



Task 1: Implementation (up to 10 marks)

You must implement a genetic algorithm that learns appropriate mapping function from creature's percepts to actions, which govern its behaviour in the simulated world. The engine that simulates the world is provided. You can choose to work in Java or Python.

The environment

The world implemented by the engine is an $S \times S$ square grid. At the beginning of the simulation the creatures, monsters and strawberries are placed randomly on the grid. Each

creature starts with a certain amount of energy. The simulation runs for T turns. After each turn, a creature loses some energy. Once the energy goes to zero, the creature dies. In order to gain more energy a creature needs to eat strawberries. When a creature finds itself on the same space as a strawberry it can eat it.

The life cycle of strawberries goes from green state to red. Eating a red strawberry gives the creature some extra energy. Eating a green strawberry provides a bit of energy as well, but less than that of the red one, plus a sore tummy, which renders the creature sick (and immobile for the next 3 turns). Once eaten, strawberries will re-spawn at random locations after some number of turns.

The creatures need to avoid monsters, which move randomly, but are capable of detecting a creature in its immediate neighbourhood and of giving a chase. If a monster moves to the same square as the creature, the creature gets eaten and it remains dead for the reminder of the simulation. Monsters move every other turn (with some moving on the odd and others on the even turns), while creatures (unless sick after eating a green strawberry) can move on every turn. Monsters can only move up, left, down, or right, while creatures can also move diagonally. Thus, creatures move faster and have more movement options than monsters.

An implementation of the world is provided along with a visualisation. You can control the size of the grid, S, and the number of turns, T, for your simulation. The number of monsters, creatures and strawberries is a proportion related to the world size. Before the start of the simulation, you will need to provide a new population of creatures. The behaviour of the creatures will be governed by the agent function that you will need to implement.

The world grid wraps around from left to right and from top to bottom. This way there are no walls. You will notice, when looking at the animation of the simulation, that creatures and monsters sometimes move past the edge of the world, disappear and emerge on the other side. Neither the creatures nor monsters perceive the edges - they just sense the neighbouring cells, which might be the ones wrapped around on the other side of the visualised grid.

The agent function

The behaviour of a creature is given by the agent function, which produces an action vector in response to a percept vector. On each turn of the simulation, each creature's agent function will be executed. A creature, or rather its agent function, receives a percept vector with information about its immediate neighbourhood. The output of the function specifies the action vector that the world interprets as what the creature wants to do. The percepts inform the creature about the presence of monsters, other creatures, or food in the eight neighbouring cells around its current position. They also contain information about the presence of the strawberry on the currently occupied square, and whether it's

green or red. There can be no other creature on the currently occupied square, nor there can be a monster (since it would have eaten the creature, not giving it a chance to execute its agent function). You will implement the agent function by encoding a model that maps the percepts to actions. The choice of what model to use is yours. However, the model must be parametrised by a set of parameters that you encode in a chromosome, so that the percepts to action mapping (i.e. the creature's behaviour) can be modified by changing the chromosome (i.e. the parameters of the model).

Evolution

Evolution of your creatures' behaviour will be done over a number of simulations of the world. You will start by creating a population of creatures (the engine will tell you how many are needed for the chosen world grid size) with random chromosomes, which should give rise to different random behaviours as dictated by the agent function. After running the simulation for T turns, you will be able to extract information from the population of your creatures - how much energy they have, time of death (if died), etc. This information should be sufficient for you to create a fitness function - a way to score the degree of success for each creature in a single simulation. Then, using principles of parent selection, cross-over, mutation, and/or elitism you will create a new generation of creatures, with different values of the chromosomes and different behaviour as dictated by the agent function. Then you run the simulation again and evaluate the behaviour of the next generation. This process can continue over a number of generations, where each creature in a given generation is evaluated over T turns of the simulation. The aim is for the creatures to eventually develop intelligent behaviour that allow them to survive for longer. The engine provides visualisations of the simulation in the environment, so that you can inspect creatures' behaviour before and after the evolution.

Percepts

The percept vector consists of 9 attributes, which give information about the nine neighbouring squares around the creature, allowing it to detect monsters, other creatures and food.

- x_0 to x_8 - correspond to nine visible squares. For the 8 neighbouring squares $x_j = 3$ indicates the presence of a food, $x_j = 2$ the presence of other creature, $x_j = 1$ the presence of a monster, and $x_j = 0$ indicates that that square is empty. If a square contains a strawberry with a monster, the presence of the monster is reported. If a square contains a strawberry with a creature, the presence of the creature is reported. For the x_j corresponding to the square that the creature is standing on, $x_j = 2$ indicates the presence of red strawberry, $x_j = 1$ a green one, and $x_j = 0$ no food.

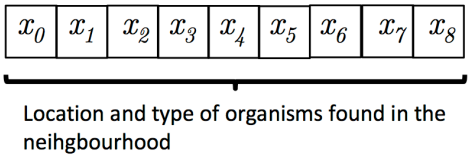


Figure 1: Percepts format 3.

The percept format doesn't provide the creature with information whether the strawberry on the neighbouring cell is green or red (but it does provide this information when food is found on the cell the creature is standing on).

Actions

The output of the agent function must consist of an array of 11 numbers, each corresponding to a weight of a different action. The engine will chose the action corresponding to the largest number in the set of 11. So, the absolute values of the action vector are of no importance – it's the relative values that correspond to the decision which of the 11 actions is taken.

- y_0 to y_8 - the first 9 numbers correspond to 9 possible choices of movement: up, down, left, right, up-left, up-right, down-left, down-right and staying in the same square. The order of this actions given here is not necessarily the order of the actions seen by the engine.
- y_9 - is related to the action of eating. If that action is taken, a strawberry will be eaten, if one is currently present on the same square that the creature is standing on
- y_{10} - is an explorative action, or a decision to make a random movement.

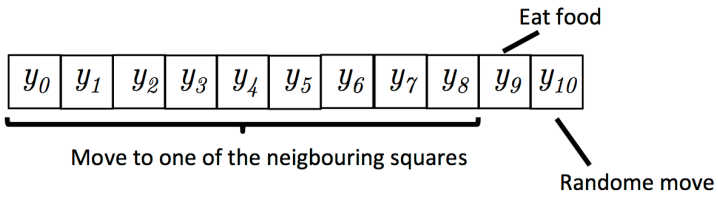


Figure 2: Actions format.

Remember, there is no invalid format for actions. Only one action can be taken by the creature and it's the one for which y_j is the largest. Your percept to actions model should discover on its own what output it needs to produce through the genetic algorithm.

World types

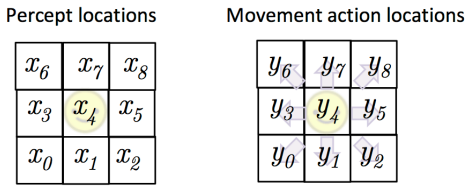


Figure 3: Correspondence of the percept and action indexes to the creature's neighbourhood for World 1.

You can choose to run you simulation in one of the two worlds. In World 1 the correspondence of the indices of the percepts, as well as the first 9 actions with respect to the location of the neighbouring attributes, is fixed as shown in Figure 3. And so, for instance, in this world the value at index 6 of the percept array always provides information about the North-West square (with respect to the creature). Similarly, the action array (as returned from your agent function) with the largest value at index 6 will always be interpreted as the movement to the North-West square (with respect to the creature). In World 2 the correspondence of the percept indexes and the first 9 action indexes to the square locations is chosen at random (when the world is created). Of course, this correspondence will remain the same for all generations of the particular simulation but it will change each time you run your program. This means that your agent function mapping percepts to actions cannot make assumptions about the meaning of the information at particular percept or action index and must learn this through the genetic algorithm. Note that the meaning of actions at index 9 and 10 always remain the same in both worlds – only the order of actions with respect to movement (the first 9 indices) is scrambled for World 2.

From the learning point of view, World 1 is somewhat less challenging than World 2. In World 1 it is possible to predefine higher level behaviours (such as move away, or go towards) and build a model that picks appropriate action based on these behaviours. In World 2 everything, including the correspondence of the percepts and the movement actions with respect to the location of the square has to be learned. The model and the learning that works for World 1 might not necessarily work for World 2, whereas the learning algorithms and the model that can cope with World 2 should do just fine in World 1. Agents that work in World 1 can get the maximum of 7 (out of 10) marks for this task. Agents that work in World 2 can get the maximum of 10 (out of 10) marks for this task.

Fitness function

You need to decide how you measure creature's fitness. After the simulation of the environment finishes its number of turns for a given population, you can examine the creatures that participated in the simulation, and gather information about their state. You get the energy level at the end of simulation (0 if starved), whether the creature is dead or not, and its time of death (in turns, if not alive). A creature eaten by a monster will still show non-zero energy, which correspond to the energy at the time of death. From this you have to formulate some kind of fitness function, that will score creature's performance. You also need to record the average fitness of the creatures after each generation and display it as a graph in your report. It's one of the things that will tell me whether you genetic algorithm works in the sense that the creature's behaviour is improving over generations.

New generation

You can use the set of creatures from the last simulation and use them as parent of a new generation that will be tested by the next simulation. You must create a population of creatures of the same size as that used in the last simulation. Using a creature from old simulation in the next simulation is allowed (that's what you do in elitism) - the engine will reset creature's state to alive and give it its starting energy again.

How you decide to implement the selection of parents, cross-over operation that produces children, mutation, and any other aspect of genetic algorithm, is up to you. However, you will have to justify your choices in your report. Remember, you need to think how the chromosomes relate to the state of your percepts-to-actions model and what kind of cross-over makes sense. A big challenge of this assignment is to design a model and cross-over operation such that two fit parents have a good chance of producing a fit child.

Repeatability

The nature of the simulation of the creatures' world is such that the element of chance does play a role in the odds of a creature's survival. It might just happen that a fit creature happens to be unlucky in a simulation, and dies early even though, if given a chance, it would display a good behaviour. While this is something that your genetic algorithm should be able to work with in the final submission, it might help your development to run simulations with consistent starting conditions. Hence, for the purpose of development, you can set the world to start a simulation for ever generations from the same state (i.e. the initial location of creatures and monsters and food). This will make the engine use the same random seed, thus resulting in same initial conditions. Of course, the final state of the simulation might not be exactly the same, because different behaviour of creatures might lead to different outcomes. But at least you will a have something more deterministic to work with in order to evaluate creatures' fitness. Repeatability will be turned off for the

final demonstration of the genetic algorithm, so make sure that at some point you run your evolution without it.

Demonstration

The code that you'll be submitting must do the following:

1. Create a world of $S \times S$ grid size. The choice of the size is yours, but don't make it too big nor too small. The default size is $S = 24$.
2. Start with a population with random chromosomes and random behaviour
3. Run the world simulation over T turns. The choice of the T value is yours, but don't make it overly long. The default value for $T = 100$. The creatures start with enough energy to survive to $T = 50$ if they don't eat at all and don't get eaten by a monster.
4. Show a visualisation of the simulation (visualisation provided by the engine) with creatures displaying non-intelligent behaviour.
5. Run evolution over number of generations. Try to design your system so this doesn't take too long. I have quite a lot of demos to run and I can't wait for half an hour for yours to finish. No need to show visualisation of the behaviour after each generation, but do show (as a graph, or just print out) the average fitness after each generation, so I can see whether the creatures are improving.
6. Show a visualisation of the simulation of the final generation. I will be looking for behaviours such as avoiding monsters and eating food.

You will not be demonstrating the code in person - I will run the code on my machine. Hence, your code must compile (if Java) and run without errors. I will not have time to debug your program. Sticking to the framework provided is the best way to assure it will run on my machine.

The engine

A framework project with the world simulation engine + visualisations is provided in Java and Python. Chose whichever framework you prefer. For files and instruction on how to use a particular engine see the "How to use the Java World Simulation Engine for Assignment 2" or the "How to use the Python World Simulation Engine for Assignment 2" page on Blackboard in the Assignments section.

You can instruct the engine whether to run a World 1 or World 2 simulation. Make sure to specify the world that you want to be marked on for your final submission. If you decided

to work with World 2 and your genetic algorithms doesn't work well, I will not run the simulations in World 1. It is probably a good idea to start working in World 1. Once you get something working relatively well in World 1, save your work, and try to adapt your genetic algorithm to work in World 2. If you can't get it to work for World 2, submit your assignment configured for World 1. If you simulation works for World 2, I will assume it works just as well for World 1.

Final note about the implementation

Remember, the point of this assignment is not to design an algorithm for successful behaviour, but rather to design a model that can learn appropriate behaviours through genetic algorithm. Encoding intelligent behaviours yourself into the agent function will not gain you many marks. Your model must be such that, when initialised with random parameters, there is little chance of creatures doing well. Creating a model capable of learning appropriate behaviour through genetic algorithm is what this assignment is about.

Task 2: Report (10 marks)

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

- A description of your simulation. This just needs to specify and explain choices of the grid size, number of turns per simulation, the world type etc.
- A description and reasoning for the choice of the model of the agent function.
- A description and reasoning for the choice of the chromosome that governs the model parameters.
- A graph showing how average fitness of the population changes as evolution proceeds.
- A description of your genetic algorithm - the method and parameters of selection, cross-over and mutation.
- A discussion of the results and how the evolution shaped your creatures' behaviour.

Marking scheme (Total: 20 marks)

This is an individual assignment.

Marks will be allocated as follows:

- Task 1: **up to 10 marks**. This task will be assessed by running your code. I'm looking for the following things:
  - The animation (with the provided visualiser) of your world showing the random behaviour of the initial population, before any evolution has happened;
  - The animation (with the provided visualiser) of the world showing the behaviour of the final population in your simulation, after several generations of evolution.

  If your evolution takes longer than 5min to run, it might be a good idea to provide a video of your simulation...or perhaps implement a feature that allows you to save and restore the state of the final population to quickly jump to the final result. In either case, I should be able to run your code in the full evolution with some indication of improving fitness to convince myself that your genetic algorithm can get on its own to the final state.

  I am looking for evidence of the genetic algorithm working – good behaviours being learned (not statically encoded), average fitness improving, successful survival strategies evolving through your genetic algorithm.

- Task 2: **10 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** (**14 May**). You should submit via Blackboard a zipped file containing your code and the report, by 4pm that day. Please make sure that you submit a working project, so that I can run run it on my machine. Best way to test it is to make sure it runs on macOS in the lab (using cosc343 virtual environment if using python). Your report should be in the PDF format.

You will lose 10% per day of available marks for late submission.