COSC343 Assignment 2

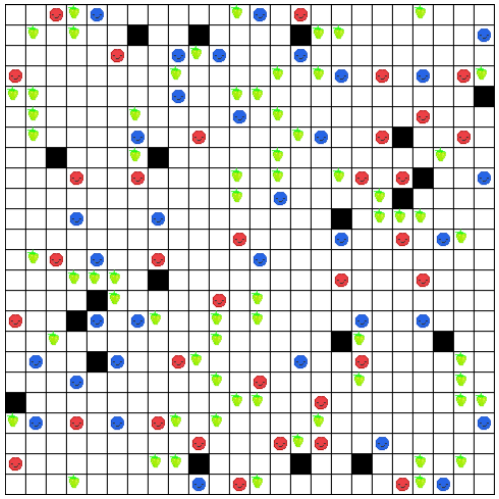
Evolve a species in a game

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 2D grid-based game. The game pits two populations of creatures (red and blue) against each other – one population controlled by an agent you are tasked with implementing. The objective is for the agent to eat strawberries, placed at random on the grid, grow in size and eat opposing creatures of smaller size. The creatures grow in size by eating strawberries or opponent's creatures. Your algorithm should find behaviours that lead to more of your creatures (regardless of size) surviving at the end of the game. The trick is that each creature acts as an independent agent and these behaviours must be learned through the process of simulated fitness selection and evolution – i.e. a genetic algorithm.

Task 1: Implementation (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 implements the game world is provided in Python.



The environment

The game takes place on an $S \times S$ square grid. At the beginning of the game the creatures, strawberries and walls are placed on the grid (symmetrically, so that neither player has a starting advantage). Each creature starts with energy $e = 2$. A game runs for T turns,

during each creatures can perform various actions. After each turn, the size of the creature is $s = \lfloor \log_2 e \rfloor$. This means creatures start with size $s = 1$, increase to $s = 2$ upon gaining $e = 4$, and to size $s = 3$ after gaining $e = 8$, etc. In order to gain more energy a creature needs to eat strawberries and/or creatures of the opposing player.

Creatures can move to neighbouring squares going left or right, up or down. Creatures can't move onto a wall and they can't move onto a space occupied by another creature. When a creature finds itself on the same space as a strawberry it can eat it to gain 1 unit of energy.

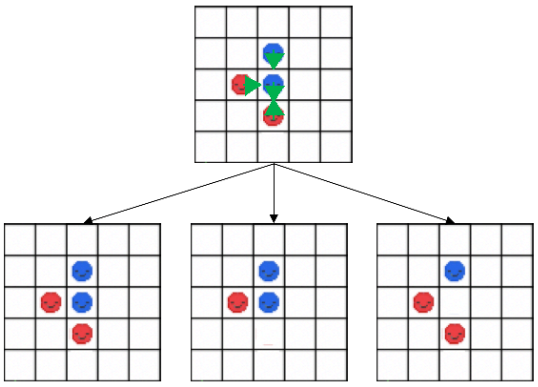
If a creature performs a movement action onto a space currently occupied by another creature, this action is taken as an attack (if the target creature is of the opposite colour) or support (if the targets creature is of the same colour). The attack is successful if the sum of energies of all attackers is larger than the sum of energies of all defenders (the attacked creature counts as a defender), in which case the attacked creature becomes "dead", is removed from the board, and its energy is divided equally among the attackers. The energy of the defenders (other than that of the target creature) is not affected by a successful attack. If attack is not successful, the attacked creature remains in play. Attacks are resolved before other actions, and so a creature can get attacked even its action is to move off its current square. Attackers and supporting defenders, regardless of the outcome of attack, remain in their square, even if the space of the attacked creature is vacated (due do the creature "dying" or moving off after "defending" against the attack). Figure 1 shows several different attack scenarios to further clarify these mechanics.

The creatures need to eat and avoid being eaten. Strawberries are a finite resource that doesn't replenish over the course of T turns (that is not until a new game, with new population of creatures starts). A match between two players, is scored as $N - M$ where N is the number of your creatures and M is the number of opponents creatures – you get positive points for more surviving creatures and negative points otherwise (0 in case of a tie). The match winner is the player with more creatures after last T of the game (for purposes of match scoring, size of the survivors doesn't count).

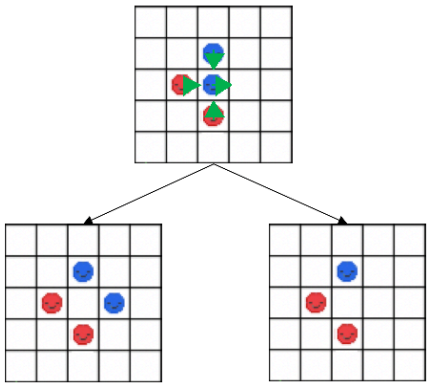
When two (or more) creatures' actions destine them for the same unoccupied square, they bounce (regardless of size) and stay where there were.

An implementation of the game is provided along with a visualisation. The behaviour of your creatures will be governed by the agent function that you will need to implement.

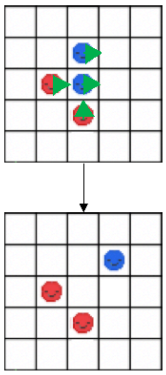
The game grid wraps around from left to right and from top to bottom. You will notice, when looking at the animation of the simulation, that creatures and monsters sometimes move past the edge of the world, disappear and emerge on the other side. The creatures do not perceive grid edges - they just sense the neighbouring cells, which might be the ones wrapped around on the other side of the visualised grid.



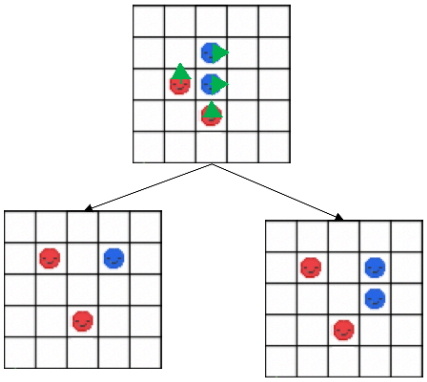
(a) Reds attack lower blue, upper blue defends lower blue, lower blue attacks lower red: every creatures is involved in attack/defence, so no one will move, bottom red might "die" from lower blue's attack, bottom blue might "die" from reds' attack; theoretically it's possible for both red and blue to die, but given possible energies for size of creatures involved, in this case this will never happen.



(b) Reds attack lower blue, upper blue defends lower blue, lower blue pick a move into new spot action; either the attack is defended and lower blue gets to execute its move action, or the attack is successful and lower blue "dies".



(c) Reds attack lower blue, lower blue is not supported; lower blue "dies" because with $s = 1$ it cannot have more energy than two red creatures of the same size.



(d) Bottom red attacks lower blue; blue either "dies" or it has enough energy to defend the attack.

Figure 1: Four scenarios with different choices of actions and possible resulting resolutions; actions are shown as green arrows, possible resolutions are shown underneath as possible scenarios; all the creatures involved are of size $s = 1$, therefore each can have either $e = 2$ or $e = 3$ units of energy.

Game parameters

You can change game parameters (grid size, number of agents, games played, etc). However, for the purposes of marking, the games will be run on the 24x24 grid, 20 walls, 34 creatures per player, 100 turns per game.

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 walls, other creatures (and their sizes), and strawberries in the 24 neighbouring cells around its current position. They also contain information about the presence of the strawberry on the currently occupied square as well as the acting creature's size. Creatures don't receive information about their own or other creatures' energy levels. 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 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).

Percepts

The percept will be given as a 5x5x3 tensor, labelled x in this description, which corresponds to three 5x5 maps of creatures surroundings with information about creatures, strawberries and walls in the neighbourhood.

- $x_{:,:,0}$ is a 5x5 map with information about neighbouring creatures
  - $x_{2,2,0}$ corresponds to the square where the creature is located and it contains a positive integer specifying this creature's size
  - $x_{i,j,0}$ specifies the relative square in the neighbourhood of 24 squares around center with i indexing the row and j the column; $x_{i,j,0} = 0$ means square i, j contains no creature, $x_{i,j,0} < 0$ means there is an enemy at square i, j, and $x_{i,j,0} > 0$ means there is a friendly creature at square i, j; absolute value of $x_{i,j,0}$ relates the creature's size
- $x_{:,:,1}$ is a 5x5 map with information about neighbouring strawberries

- $x_{i,j,1} = 1$ indicates a strawberry at square i, j, otherwise $x_{i,j,1} = 0$; note that $x_{2,2,1} = 1$ indicates a strawberry on the square the creature is currently on

- $x_{:,:,2}$ is a 5x5 map with information about walls
  - $x_{i,j,2} = 1$ indicates a wall at square i, j, otherwise $x_{i,j,2} = 0$

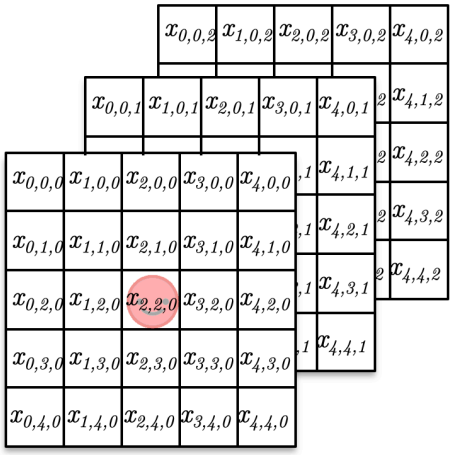


Figure 2: Percepts format.

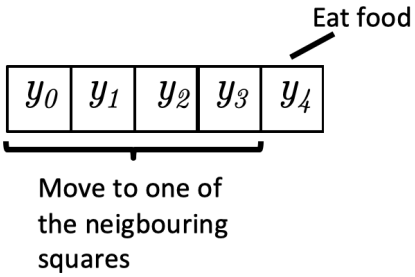
The percept format doesn't provide the creature with information about the energy of any creatures – just the size.

Actions

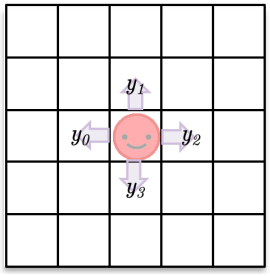
The output of the agent function must consist of an array of 5 numbers, each corresponding to a weight of a different action. The engine will choose the action corresponding to the index of the largest number in the set of 5. So, the absolute values of the action vector are of no importance – it's the relative values that correspond to the decision which of the 5 actions is taken.

- y_0 to y_3 - the first 4 numbers correspond to 4 possible choices of movement: left, up, right, and down respectively.

- y_4 - is related to the action of eating. If that action is taken, and the creature is on a square with a strawberry, that strawberry will be eaten



(a) Correspondence of between the action vector indexes and the actions taken



(b) Map of movement actions

Figure 3: Actions format.

Remember, as long as action is a list of 5 numbers, 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.

Training

On its first run you agent gets trained by playing against a selection of opponents (specified by you, more about this later on). You agent should learn through evolutionary process over the course of up to 1000 games/generations. Your creatures should initialise with random chromosomes, which should give rise to different random behaviours as dictated by the agent function. In the first turn of the training your creatures should (on average) not do better than random player (provided along with the engine files). After running the simulation for T turns, you can extract information from the population of your creatures that just played the game - their size, strawberries eaten, 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 should create a new generation of creatures, with different values of the chromosomes and different behaviour as dictated by the agent function. Then you a game is played again and at the end the behaviour needs to be evaluated for the next generation. The aim is for the creatures to eventually develop intelligent behaviour that allow them to survive for longer than opponent's creatures. The engine provides visualisations of the games, so that you can inspect creatures' behaviour after the evolution.

Fitness function

You need to decide how you measure creature's fitness. After a game finishes (terminated either after T turns or when one player loses all the creatures) you can examine the creatures that participated in the game, and gather information about their state. You get their size at the end of the game, whether the creature is dead or not, its time of death (in turns, if not alive), count of strawberries eaten, total energy gained from attacking enemy creatures, number of bounces (unsuccessful moves onto a square where other creature attempted to move at the same time) and total number of squares visited. From this you have to formulate a 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 game and use them as parents of a new generation that will be tested by the next game. You must create a population of creatures of the same size as that used in the last game. 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 size and 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.

Opponents

The engine provides two non-evolving players - random action player (the name explains how it works) and a hunter player (with few basic fixed behaviours that make this a reasonably strong, but not trainable, player). These are provided so that you've got players to train your agent against. For the training regime of your agent you can specify training sessions against random, hunter or self (your agent playing itself) in arbitrary order for arbitrary number of generations/games for up to 1000 games in total. If you have no idea what is appropriate for your training schedule, train against random player. Also, remember – random and hunter players have fixed behaviours, which don't change over generations. Hence, while their code might serve as an inspiration, don't emulate it too much – it's missing the learning models and the genetic algorithm learning, which constitute the most important aspect of this assignment.

Demonstration

The code that you'll be submitting will be a single file, `myAgent.py`, for which the boiler plate is provided with the engine. This code must specify:

1. `MyCreate` class that initialises with random values chromosome and contains the `AgentFunction` that implements a model parametrised by the chromosome mapping percepts to returned vector of actions
2. `newGeneration` function that takes a list of `MyCreature` objects and returns a new list of `MyCreature` that constitutes a new generation of creatures
3. A list of tuples specifying the trainingSchedule - for instance, to train for 500 games against random player and then for another 200 against self, the schedule would be `[('random',500),('self',200)]`.

You will not be demonstrating the code in person - I will run the code on my machine. Hence, your code must 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 game simulation engine + visualisations is provided in Python. For files and instruction on how to use a the engine see the "How to use the Game Simulation Engine for Assignment 2" section on Blackboard in the Assignments section.

Tournament

A server running continuous tournament has been set up at `http://jet343.otago.ac.nz`. You can login with your cs username using your student number as the password. You can submit your version of myAgent.py, which will get trained according to your schedule and start playing games against other submissions. At the end of this assignment we will run the final tournament to determine the winner. For more information on the tournament server refer to "How to use the Tournament Server" section on Blackboard in the Assignments section.

Final note about the implementation

Remember, the point of this assignment is not to design a FIXED 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, it has little chance (at first) of having the creatures doing well. I will test your creatures against the random player. Your creatures should do no better (on average) than random player in their first training game, but decisively beat random player after training. If your creatures are consistently winning in game 1 of the training against the random player, that means your agent's behaviour is hard-coded into the combination of chromosome/model and not learned through GA.

Task 2: Report (10 marks)

You must also write a report about your implementation of the creature and the agent function. The report should include:

- 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: **10 marks**. This task will be assessed by running your code and inspecting behaviours through visualisation.

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

  Comment your code well and clean it up before submission (no debugging/commented out code)

- 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** (**18 May**). You should submit your entire code contained in a single file (`myAgent.py`) and report in PDF format via Blackboard.

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

Academic Integrity and Academic Misconduct

Academic integrity means being honest in your studying and assessments. It is the basis for ethical decision-making and behaviour in an academic context. Academic integrity is informed by the values of honesty, trust, responsibility, fairness, respect and courage. Students are expected to be aware of, and act in accordance with, the University's Academic Integrity Policy.

Academic Misconduct, such as plagiarism or cheating, is a breach of Academic Integrity and is taken very seriously by the University. Types of misconduct include plagiarism, copying, unauthorised collaboration, taking unauthorised material into a test or exam, impersonation, and assisting someone else's misconduct. A more extensive list of the types of academic misconduct and associated processes and penalties is available in the University's Student Academic Misconduct Procedures.

It is your responsibility to be aware of and use acceptable academic practices when completing your assessments. To access the information in the Academic Integrity Policy and learn more, please visit the University's Academic Integrity website or ask at the Student Learning Centre or Library. If you have any questions, ask your lecturer.

- Academic Integrity Policy
- Student Academic Misconduct Procedures