AIML402 Assignment 2

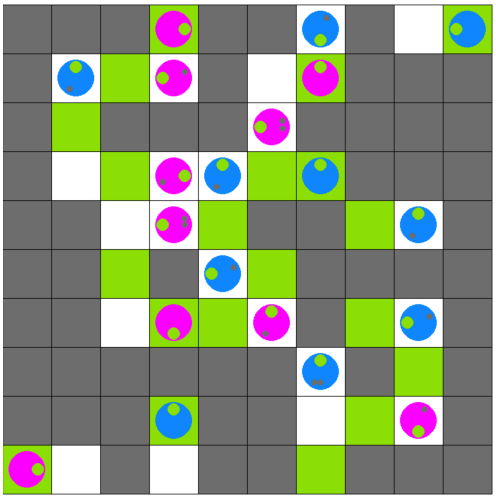
Cleaners!

Weight:10% Lecturer: Lech Szymanski

For this assignment, you will implement a genetic algorithm to optimise the fitness of a population of vacuuming agents, a.k.a. cleaners, tasked with cleaning an area. The rules of the world are as follows: a cleaner can move forwards on backwards in discrete steps; it can rotate in either direction by 90 degrees; when driving over dirty area it automatically picks up the dirt (unless its bin is full); cleaners run on a battery that needs to be recharged periodically at a charging station, where also the bin is emptied. The objective is clean maximal area. Oh, and the final complication is that there is another population of cleaners on the field, competing for cleaning credits. Your algorithm should find behaviours that lead to the most cleaning done by your cleaners. The trick is that each cleaner in your populations 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 (10 marks)

You must implement a genetic algorithm that learns appropriate mapping function from cleaner's percepts to actions, which govern its behaviour in the simulated world. The engine that implements the game world is provided in Python.



The environment

The game takes place on an $S \times S$ square grid/plane. At the beginning of the game, a population of cleaners is placed on the field at random locations, each starting on a square

with a charging station (and thus the full battery charge). A population of opposing cleaners is placed symmetrically, with the starting positions reflected about the diagonal going from left-bottom to right-top corner. The game runs for N turns, during which each cleaner, provided its battery is not depleted, can perform a single action: either go forward or backward, or turn 90 degrees to the left or right. Each turn the cleaner loses one unit of charge.

When a cleaner goes over dirty area and its bin is not full, it will automatically suck up the dirt. When a cleaner visits the charging station, its battery is immediately restored to full charge and its dust bin is automatically emptied. When a cleaner runs out of charge, it can't perform any actions. Full battery allows the cleaner to get to just about half of the width/height (whichever is smaller) of the field.

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 cleaners sometimes move past the edge of the world, disappear and emerge on the other side. The cleaners 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.

The engine for the assignment is provided along with a visualisation of the game. The behaviour of your cleaners will be governed by the agent function (that you will need to implement) that is to be parameterised by a chromosome, so that different chromosome values lead to different behaviours.

You can change game parameters (grid size, number of cleaners, number of turns per game, etc) for the purpose of debugging and development. However, for the purposes of marking, the game will be run on the 41x41 grid, 40 cleaners per player, 100 turns per game.

The agent function

The behaviour of a cleaner is given by the agent function, which produces an action that moves the agent either forwards, backward, or rotates to left/right by 90 degrees. Every turn every agent that has enough energy for an action gets to executed an action. A cleaner, or rather its agent function, receives a tuple of percepts with information about its immediate neighbourhood. The output of the agent function must be a vector of four values which will be interpreted as weightings of one of 4 possible actions. The percepts inform the cleaner about the dirty/clean state of its surroundings, location of the energy/empty stations and presence of other cleaners. Any cleaner can use any energy/empty station. You will implement the agent function that 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 chromosome, so that the percepts to actions mapping (i.e. the cleaner's behaviour) is different for different values of the cleaners's chromosome.

Percepts

The percepts will be given as a tuple that contains visual information, state of the battery, state of the dust bin and state of the success of previous actions. The visual field aspect is a $3 \times 5 \times 4$ tensor, which provides information about the 3×5 grid in the cleaner's immediate surroundings. The cleaner's field of vision allows it to only *see* forward and to the sides.

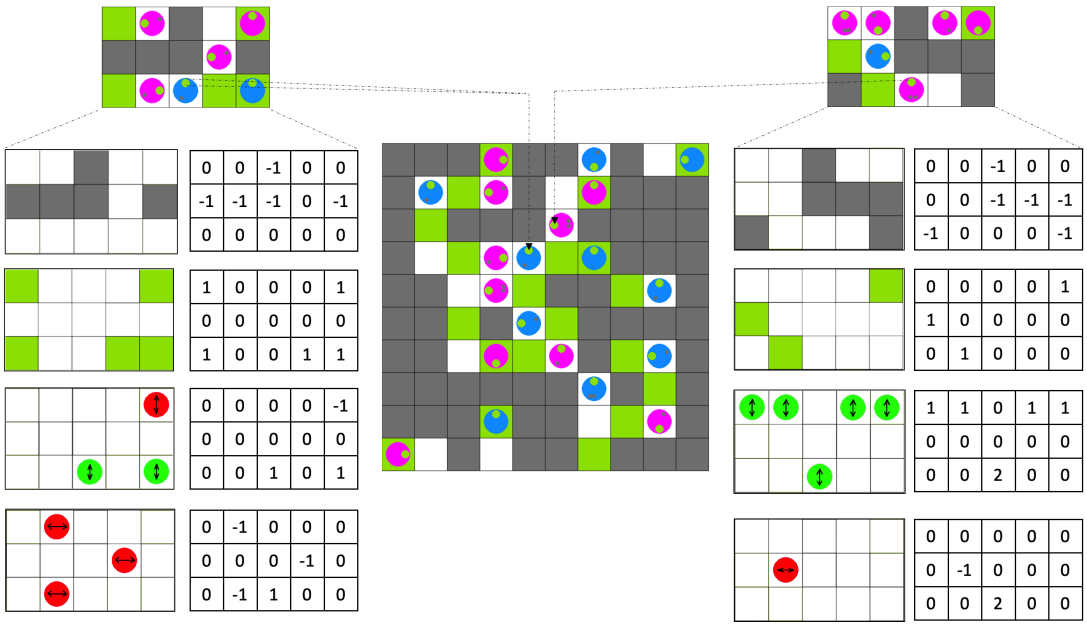


Figure 1: Visual percepts for two cleaners in the above situation (middle image); the left side shows the visual field of the blue cleaner (pointed to by the arrow) oriented up, with the visual field showing in its vicinity: 5 dirt spots, 5 charge stations, 3 cleaners oriented along the same axis (2 friendly, including the cleaner itself, one enemy) and three cleaners oriented along 90/270 degrees to this one (all enemy); the right side shows the visual field of the purple cleaner (pointed by the arrow) oriented to the left, with the visual field showing in its vicinity: 6 dirt spots, 3 charge stations, 5 friendly cleaners oriented along the same axis (including the cleaner itself), and 1 enemy cleaner at 90/270 degrees to this one (note that the cleaner's field of view is rotated so the locations of the visual percepts are always in the same position related to the cleaner).

The percept matrix contains information about the squares in the grid around given cleaner as well as energy, bin and last actions failure states. The visual field allows the cleaner to *see* a 3x5 grid around the cleaner (two columns to left and right, two rows forward not counting the column/row where the cleaner resides). There are different aspects of the vision so there are 3x5 visual maps. In each map, the cleaner is located at the bottom

middle square facing up. The orientation of the visual field is given from the cleaner's point of view. The first map indicates locations of dirt in the cleaner's field of view (denoted with -1's), second map shows locations of charge stations (denoted with 1's), third map shows cleaners that face either up or backwards in relation to this cleaner (enemies denoted with -1's and friendlies with 1's), and the fourth map shows cleaners facing left or right in relation to this cleaner (enemies denoted with -1's and friendlies with 1's). A bot does not distinguish between other bots' facing up or down, since those bots can move either up or down regardless whether they face up or down (from the seer's point of view). Similarly, a bot does not distinguish other bots' facing left or right, since those bots can move either left or right regardless whether they face left or right (from the seer's point of view). This is a simple consequence of the fact that each bots is allowed to move forwards or backwards. See Figure 1 for an illustration.

Aside from the visual percepts, the cleaner also receives information about the charge left in the batter (how many turns it can remain active without recharging), the state of its dust bin (the number of squares it can still clean without emptying the dust bin) and the state of success of its last action(s).

Actions

The output of the agent function must be a 4-value vector, with the values interpreted as the weighting of the desire to go forward, turn right, turn left or go backwards. For example, if the first value is the largest, then action is forward, if the second value is the largest, the bot will turn right. One of the actions must be taken. Turning always succeeds. Going forward or backward might fail if it results in a collision with other cleaner. Collisions are resolved by the engine and actions that result in a collision don't succeed - neither of the agents gets to go forward or backward.

Population of cleaners

You may wonder how one agent function (that you are meant to implement) can control an entire population of cleaners. Wouldn't all cleaners behave the same? Not necessarily. First of all, different cleaners are likely to receive different percepts at different time, depending what is going on around them. But more importantly, the agent function needs to depend on values of the chromosome. In the engine (provided for this assignment) you will implement a Python class that can be instantiated multiple times. In the __init__ method of that class you should initialise the chromosome (in whatever format the model of your agent function needs) with some random values. That chromosome variable will be unique to a given instance of the object of that class. The engine will instantiate K objects of that class, which correspond to K cleaners, each with a different copy of the chromosome (possibly saved in its self.chormosome variable). The engine tracks the location of each cleaner throughout the game and executes the agent function on the instance of the object

providing appropriate percepts. Inside the agent function, the decision of whether to return a 4-dim vector that will send the cleaner forward/backward, left or right should depend not only on the value of the percepts but also on the values of the chromosome (i.e. the self.chromosome, if that's what you choose to name it), which should be different for each cleaner instance. And voila - different behaviour of different cleaners. Of course, the objective is to evolve the chromosome by process of GA, so that cleaners with better behaviours are more likely to produce children, which cross-over their parent chromosomes and (hopefully) get better at cleaning without running out of charge.

Training

The chromosome of your cleaners should be such that when the game is run on randomly initialised population (i.e. of cleaners with random chromosome values), the cleaners in the population should exhibit different individual behaviours. In addition to the agent function, you need to implement a GA algorithm that searches through the space of chromosomes for values that give rise to "desired" behaviours in your agent function, as measured by fitness function that you are to implement as well.

For the purpose of training, the engine will create a population of random cleaners at the start, and then run the game for N turns, giving chance to the cleaners to act in the world. After that, you'll have an opportunity to evaluate the success of the individuals in the last game and produce a new population for the next generation, based on the stats provided about each cleaner.

After you evaluate the fitness of the clenaers, you need to create a new generation using GA principles of parent selection, cross-over, mutation, and/or elitism. The new generation does not carry random chromosomes anymore, but cross-overs of various parent pairs (selected based on fitness) with a touch of randomness in the way the cross-over combines parent's states and possibly some chance of random mutation.

Then, the game is played again for the next generation, which you later can evaluate for fitness again. The GA process can continue for up to 500 generations. The aim is for the cleaners to eventually develop intelligent behaviour that allows them to clean more, as the final score for the game is the total number of squares cleaned by your agents minus the total number of squares cleaned by the opponent. You want a positive score, and the larger the better.

Fitness function

You need to decide how you measure cleaner' fitness. All cleaners start on the charging station with a full battery and an empty bin. After one run of the game finishes (terminated either after N turns or when all cleaners have drained their batteries) you can examine the cleaners that participated in the game, and gather information about their state. For

each cleaner, you'll have information about its performance as a dictionary of stats from the game:

- `cleaned` – total number of dirt loads picked up
- `emptied` – total number of dirt loads emptied at a charge station
- `active_turns` – total number of turns the agent was active (non-zero energy)
- `successful_actions` – total number of successful actions performed during active turns
- `recharge_count` – number of turns spent at a charging station
- `recharge_energy` – total energy gained from the charging station
- `visits` – total number of squares visited

This information should be sufficient for you to create a fitness function - a way to score the degree of success for each cleaner in a single game.

An an example of one possible fitness function is:

$$f = \#\text{cleaned}. \tag{1}$$

How good is the fitness in Equation 1? Well, it only values cleaning, so two cleaners that picked up two dirt loads, but one died half way through while the other kept recharging (but not picking up more dirt) are deemed to be equally fit. This might be an ok for fitness, but possibly not capturing everything that it should taken into account. At the same time, it does provide an implicit reward to cleaners that recharge, since they have more opportunities to pick up dirt. You are free to use this fitness function, though augmenting it might be a better way to go.

Model of the agent function

If you absolutely have no idea how to even get started thinking about the model, here's a working example of how to use a linear/perceptron-like model. If we reshape the $3 \times 5 \times 4$ percepts tensor (here the $\times$ symbol stands for "by") to a $(3 * 5 * 4)$-dim array, and concatenate it with a 3-dim array (containing values of energy, bin state and fails) into a 63 dimension vector x, then we have an array of percepts indexed x_1 through to x_{63} (I am indexing from 1 to 63, since that's the convention in maths – you can just translate it to indexing from 0 to 62 for your program). A single output perceptron would assign a weight to each of those percepts and compute the output as

$$v = w_1 x_1 + w_2 x_2 + ... + w_{63} x_{63} + b, \tag{2}$$

where b is the bias value. By choosing random values for w's and the b you have different output for the same x, and that's your chromosome – the (64) weight values plus bias that can lead to different behaviours.

But we need four outputs, remember? increase your model parameters. Well, you can do this computation 4 times, with 4 sets of different weight and bias values to compute

$$v_j = w_{j,1}x_1 + w_{j,2}x_2 + ... + w_{j,63} + b_j, \tag{3}$$

for $j = 1, 2, 3, 4$ to obtain v_1, v_2, v_3 and v_4. Those four outputs could be your action vector – the largest v_j leads to action j. Given the same set of weights and biases (in a given individual) the relative sizes of v's might be different for different x (i.e. bot doing different things in different situations). But also, given different sets of weights and biases (in different individuals) the relatives sizes of v's might be different for the same x (i.e. different bots exhibiting different behaviours.

Important to note here - even though we are using a perceptron-like model, we are not training it with the perceptron rule. GA is a stochastic search, and so, rather than training the model in a supervised way, we treat the parameter values as a state of a model. GA searches through possible states and is supposed to select the states of the model that correlate with "desired" behaviours. In the first generation, random values of the parameters/weights should make for a choice of random behaviour in the population (i.e some cleaners might go towards dirty squares, some night not), but over generations, the states in the chromosome that lead to better fitness should emerge.

New generation

You can use the set of cleaners 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 cleaners of the same size as that used in the last game. Copying a cleaner from old population to the new one is allowed (that's what you do in elitism) – the engine will reset cleaners's state, give it full charge and empty bin at the start of a new game.

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 a non-evolving player - random action player (the name explains how it works). This agent is provided so that you've got a player to train your agent against. For the training regime of your agent you can specify training sessions against random, or self (playing evolving population of your cleaners against an non-evolving population of cleaners copied from the first generation of self-play) in arbitrary order for arbitrary number of generations/games up to 500 games in total. If you have no idea what is appropriate for your training schedule, train against random player.

Demonstration

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

1. `trainSchedule` variable that is a list of tuples specifying the training schedule for evolution - for instance, to train for 300 games against self and then for another 200 against random agent, the schedule would be `[('self',500),('random',200)]`.
2. `Cleaner` class that initialises a random value chromosome and contains the `AgentFunction` that implements a model parametrised by the percepts and the chromosome to returned a 4-value action vector;
3. `evalFitness` function that takes a list of `Cleaner` objects and returns a corresponding list of fitness values based on your choice of evaluation function;
4. `newGeneration` function that takes a list of `Cleaner` objects (old population) and returns a tuple made up of a new list of `Cleaner` objects (that constitutes a new generation of cleaners) and average fitness of the old population;

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 Cleaners Game Engine for Assignment 2" 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 multiple time with random parameters, the behaviours for each cleaner should be different and not always conducive to the cleaner doing well. So, while the behaviour on individual level should be deterministic (a cleaner should do the same thing in the same situation every time), the behaviour on the population level should be random (cleaners should have different behaviours and there should be no bias in the population towards good behaviours). If you want to test your populations' initial behaviour, set up a match against random with the `trainingSchedule` of your agent set to `None`. This will test your untrained cleaners against the random agent. Your cleaners should not be winning those matches every time - on average they should lose as many as they win. If your cleaners, as a population, are consistently winning without training, then that means you have hard-coded the "correct" behaviours into your cleaners' model, which means these behaviours are not learned. Of course after the GA, your cleaners should decisively beat the random agent.

Task 2 (10 marks)

You must also write a report explaining the model and the GA implementation as well a results evaluating the effectiveness of your solution. The report should include:

- a brief introduction – you don't have to repeat the entire description of the game and environment given, but a short intro to what your report is about is required;
- a description of the implementation and reasoning for the choices of the model of the agent function.
- explanation how the chromosome governs the model parameters and thus the cleaners' behaviour;
- explanation how your GA is implemented, including the description of the fitness function;
- evaluation of the GA should provide at least evidence (a graph for instance) of average fitness of the population increasing over time;
- analysis of the behaviour and the GA's performance;
- brief conclusion – summary/discussion of the results/conclusions;

- citations – if needed;
- information on how to run your code – if extra libraries are required.

Screenshots of the text in your terminal and/or photos of hand-drawn diagrams are not the best way to add figures to your report. Figures and diagrams are great to have, but draw them properly (in Inkscape or PowerPoint for instance).

To give you a bit of guidance for the report structure, a LaTeXtemplate is provided (you can find it on Blackboard in the "Assignments" section under "Report template in LaTeX"). You don't have to use LaTeX to write your report – but it might be a good idea to follow the general structure provided in the template.

Finally, to pre-empt inevitable questions about the length of the report, let's say: "around 2000 words (which is roughly 4 pages)". But this is not an absolute number – more pages if you have lots of figures is not a problem.

Marking scheme

This is an individual assignment. Marks will be allocated as follows:

- Task 1: **10 marks**. This task will be assessed by running your code, inspecting behaviours through visualisations as well as playing trained population of cleaners against self and random agent. I am looking for evidence of the genetic algorithm working – good behaviours being learned (not hard-coded), average fitness improving, successful game strategies evolving through your genetic algorithm.

  Clean up the code source (as well as any debugging printing) before submission.

- Task 2: **10 marks**. Marks will be awarded for clarity of the report, good explanation of the agent function model and the GA methodology, and at least a bit of evaluation of your GA algorithm with some analysis.

The percentage of the obtained marks out of 20 will be converted to a fraction of 10% that constitutes the weighting of this assignment in AIML402.

Submission

The assignment is due at **23:59 on Tuesday, 12 Sep 2023**. You should submit via Blackboard three files: `my_agent.py`, `my_agent.tar.gz`, and the pdf file containing the report. Don't zip these files into one attachment – add them separately to the submission.

Late submissions will incur a 5% penalty per day.

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.

Use of generative software such as ChatGPT is allowed as long as it is for the purpose of aiding, not subplanting the effort of development code and/or improving the writing. If generative software is used, students must specify (in the report's appendix) how it was used and on what aspects of the assignment.

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