AI-F19
Assignment 2 (A2)
Genetic Algorithms

General Info

10 Point Individual Assignment
Due by 2 pm Wednesday October 16th (see syllabus for collaboration policy & policy regarding late submissions)

Learning Objective

This assignment satisfies learning objective 2 (LO2) as specified in the syllabus. You will apply conceptual knowledge of core AI concepts by implementing AI algorithms, analyzing existing intelligent systems (including humans), and using existing AI tools to solve problems.

Getting Help

We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask. If you find yourself stuck on something, contact us via Piazza or come by the office hours. If you can't make our office hours, let us know and we will be happy to schedule alternate times.

Submission

You should submit the deliverables on Gradescope under Assignment 2. If you encounter any difficulties during the submission process please contact us via email and also include a copy of your submission files *before the deadline*.

Deliverables

You need to submit four files named exactly as follows: part1.py, part2.py, bestStrategy.txt and [Identikey]report_A2.pdf

Overview

This assignment has two parts, both are required. The first part has you implement a basic Genetic Algorithm, the second has you apply what you have learned to a more complex environment, experiment with it, and analyse the results. Be sure to read both parts before beginning. If your code for Part 1 is general enough, you can reuse a lot of it for Part 2.

Note: The experiments in Part 2 can take a while to run. Be sure to give yourself plenty of time.

Skeleton Code

You can download the skeleton code from Canvas - “Assignment2_SkeletonCode.zip”. This assignment uses **Python3.** You may use NumPy, as well as standard libraries such as random.

Part 1 [5pts]

In this assignment, you will implement a simple genetic algorithm to maximise the number of ‘1’s in a bit string. A genome will be a bitstring. **Selection** will be performed using fitness-proportionate selection (roulette-wheel sampling). We will use single point **crossover**, and **mutation** will be characterized as flipping a bit. If you need a reminder for each of the stages, Figure 4.6 in Russell and Norvig (and the Slides) provides a good overview.

The **goal** is for the bitstring to be all ones. We will use the following fitness function to reach the goal:

f(*x*) = number of ones in *x*, where *x* is a genome of length 20.
Eg. [01001110011001101000] would have a fitness of 9.

Implementation [3pts]:
To make implementing your GA easier, we have provided skeleton code in **part1.py**. In order to complete the first part of the assignment you should implement the following functions:

- **randomGenome(*length*)** returns a random genome (bit string) of a given length.
- **makePopulation(*size, length*)** returns a new randomly created population of the specified size, represented as a list of genomes of the specified length.
- **fitness(*genome*)** returns the fitness value of a genome.
- **evaluateFitness(*population*)** returns a pair of values: the average fitness of the population as a whole and the fitness of the best individual in the population.
- **crossover(*genome1, genome2*)** returns two new genomes produced by crossing over the given genomes at a random crossover point.
- **mutate(*genome, mutationRate*)** returns a new mutated version of the given genome.
- **selectPair(*population*)** selects and returns two genomes from the given population using fitness-proportionate selection. This function should use *weightedChoice* to select each genome. *weightedChoice* is already provided and takes two arguments, the list of genomes, and a list of weights, in this case the weights can be the associated fitness for each genome.

There are tests for each of these functions at the end of the skeleton code, simply uncomment as you go to experiment with each. We can't guarantee that this program will detect all possible bugs that may exist in your code, but it should catch the most egregious ones. Once you have implemented each of the above functions, you will then implement the function below:

- **runGA(*populationSize*, *crossoverRate*, *mutationRate*, *logFile*="")** is the main GA program, which takes the population size, crossover rate (p_c), and mutation rate (p_m) as parameters. The optional *logFile* parameter is a string specifying the name of a text file in which to store the data generated by the GA for plotting purposes. When the GA terminates, this function should return the generation at which the string of all ones was found. The GA should run for 50 generations or until the string of all ones is found. If no solution is found, return None.

Your GA program should print out, on each generation cycle, the fitness of the best individual in the current population and the average fitness of the population as a whole. It should also give the user the option of recording this output data in a text file. This will enable you to plot the results of each run for easy comparison.

Here is an example of the type of output your program should produce:

```
>>> runGA(100, 0.7, 0.001, "run1.txt")
Population size: 100
Genome length: 20
Generation    0: average fitness 10.07, best fitness 15.00
Generation    1: average fitness 10.91, best fitness 15.00
Generation    2: average fitness 11.45, best fitness 16.00
Generation    3: average fitness 12.02, best fitness 16.00
...
Generation   18: average fitness 16.09, best fitness 19.00
Generation   19: average fitness 16.38, best fitness 20.00
Results saved in file run1.txt
19
```

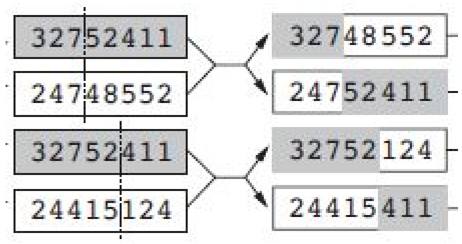
The contents of the resulting text file **run1.txt** should look something like this:

```
0 10.07 15.00
1 10.91 15.00
2 11.45 16.00
3 12.02 16.00
...
18 16.09 19.00
19 16.38 20.00
```

In your report [2pts]:

1. Perform 50 runs with the parameters listed below, measure the average generation at which the string of all ones is discovered, report this as well as the minimum and the maximum. Use the following parameters:

   population size 100,
   single-point crossover rate p_c = 0.7,
   bitwise mutation rate p_m = 0.001.

   Single-point crossover rate (p_c) is the probability that after selecting two genomes, you will perform single point crossover as shown to the right. In other words, you will only do a cross over 70% of the time.

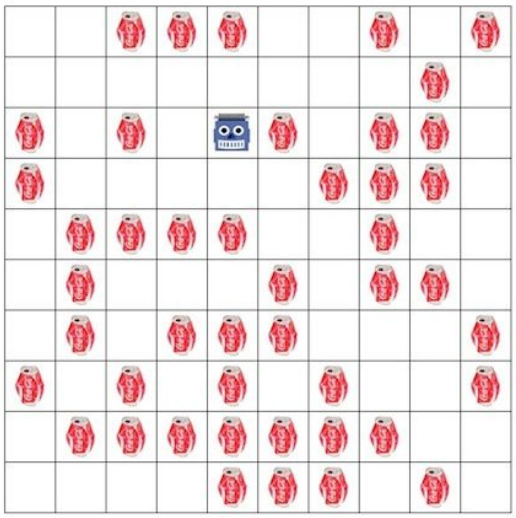
   Mutation rate tells us the probability that a bit will be flipped within the genome. For example if mutation rate was 1, all of the bits of a genome would be flipped every time.

2. Select a random 5 of the above runs and produce a line plot of average fitness of each of the five populations over time. This should result in one line plotted for each of the five runs all shown on the same graph. Generation should be on the x-axis and average fitness on the y-axis. Make sure the graph is appropriately labeled. Discuss what you notice across all five runs, note the similarities and differences.

3. Perform the same experiment with crossover turned off ($p_c = 0$).Describe the effect on the algorithm and provide an explanation.

4. Do similar experiments, systematically varying the mutation and crossover rates, and population size, to see how the variations affect the average time required for the GA to find the optimal string (if at all). Be careful to only vary one thing at a time, so that you can draw a meaningful conclusion from your simulations. In the report, write a description of your experiments. Write a brief summary (1 page max) of your experiments including which parameters produced the quickest solution and the effect of your changes on GA performance. Include data (graphics etc) to back up your claims.

Part 2 [5pts]



Now it's time to look at a more complicated environment. In this part, you will implement a genetic algorithm to evolve control strategies for Robby the Robot, as described in Chapter 9 of *Complexity: A Guided Tour* by Melanie Mitchell (available in the assignment folder on Canvas), A control strategy will be represented as a string of characters that code for the following robot actions:

- **0** = MoveNorth
- **1** = MoveSouth
- **2** = MoveEast
- **3** = MoveWest
- **4** = StayPut
- **5** = PickUpCan
- **6** = MoveRandom

Robby's reward is increased when Robby picks up a soda can, and decreased if Robby hits a wall. The job of your GA is to evolve a good control strategy that maximizes Robby's average cumulative reward as it collects empty soda cans for 200 time steps.

Your GA will maintain a population of genomes representing control strategies. Each genome will be a **243-character string** specifying the action that Robby should take for every possible situation it might find itself in. Robby's "situation" will be encoded as a 5-character *percept string* specifying what Robby currently sees in their immediate vicinity. For example, the string 'WECWC' means there is a wall to the north, an empty grid cell to the south, a soda can to the east, another wall to the west, and a soda can in Robby's own grid cell.

Each percept string corresponds to a unique code number in the range 0-242, which indicates the (0-based) index number of the situation, for more details on this check out page 132 in the chapter noted above. We will use mainly use the percept codes in our problem rather than the strings.

In this environment, there are 243 distinct situations the robot can find themselves in, each of these is represented by a percept string (e.g. ‘WECWC’) and the index of that string (e.g. 240) called the perceptCode. A **genome** is a string of length 243, comprising of numbers from 0-6 inclusive, each number represents an action and the position in the string tells us when that action should be performed (e.g. If the percept code is 240, robby will perform the action described by the 240th gene of the genome). For example, if the genome were all 0’s, in any given situation the Robot would move north.

We will use rank **selection** (described more below), single point **crossover,** and **mutation** which will be characterized as randomly replacing one or more characters in the string.

Robby's World

We have already provided a simulator for Robby’s world in the skeleton code in the folder named **robby**.

IMPORTANT: Do not put your Python file or any other files inside the folder. Just make sure the folder is in the same location as your program file. You should not modify any of the code in this folder.

You’ll note at the top of **part2.py** the following lines
```
import robby
rw = robby.World(10, 10)
```

This handles the setup of the simulator

To interact with the world, you can use the following commands, a full list of commands is shown at the end of this document. :

- **rw.getCurrentPosition()** returns the current 0-based *row*, *column* position of Robby in the grid.
- **rw.getPercept()** returns the percept string specifying what Robby currently sees to the North, South, East, West, and Here.
- **rw.getPerceptCode()** returns the code number of the current percept string as an integer in the range 0-242.
- **rw.performAction(*action*)** causes Robby to perform the specified action, where *action* is one of the strings "MoveNorth", "MoveSouth", "MoveEast", "MoveWest", "StayPut", "PickUpCan", or "MoveRandom".

- **rw.graphicsOff(*message*="")** turns off the graphics. This is useful for simulating many actions at high speed when evaluating the fitness of a strategy. The optional *message* string will be displayed while the graphics are turned off.
- **rw.demo(*strategy*, *steps*=200, *init*=0.50)** turns on the graphics and demos a strategy for the specified number of simulation steps (default 200) with a randomized soda can density of *init* (default 0.50). Optionally, *init* can be a string specifying the name of a grid world configuration file created with the save command.
- **rw.strategyM** is a string representing the hand-coded strategy *M* created by Melanie Mitchell, as described in the handout.
    - For example, you can watch strategy M in action by typing the command **rw.demo(rw.strategyM)** at the Python prompt.

Setting up your GA [2pts]

Enter all your code in part2.py, notice how the skeleton code has all the same functions.
Depending on your implementation, you may be able to reuse code from Part 1.

- **randomGenome(*length*)** should now return a string of the numbers 0-6
- **fitness(*genome*)** - This function should return the average total reward accumulated during a cleaning session when following the strategy. This should be averaged over 25 sessions. Strategies can have negative fitness.
- **Hint: rw.performAction(*action*)** returns the reward value of a certain action
- **Hint:** You may want to look at the code for rw.demo, this can get you started on how to simulate a cleaning session. For computing fitness, be sure to turn of the graphics to increase speed.
- **Hint** you may change the function parameters within your GA to improve efficiency.

For this part, you should use *rank selection* when selecting genomes for crossover and mutation. Many control strategies can have negative fitness values, which would cause problems with fitness-proportionate selection (see slides for details).

To implement rank selection, first calculate the fitness of each genome in the population. Then sort the genomes by their fitness values (whether positive or negative), in increasing order from lowest to highest fitness. The selection weight of a genome will be its rank number from 1 (lowest) to 200 (highest), instead of the fitness value itself. This will impact the **selectPair** function.

We have provided a function **sortByFitness** to aid you in rank selection. Note this is likely not the most efficient way to do this, you may want to adjust this function to improve the speed of your program.

Experimenting

We recommend using the following initial parameters for your GA:

- Population size: 100

- Crossover rate: 1.0 (meaning 100% probability of single-point crossover)
- Mutation rate: 0.005
- 200 actions per cleaning session
- Number of generations: 300

Experiments [3pts]

Run your GA for a total of 300 generations. Every 10 generations, your GA should record the best strategy from the current population, along with the strategy's fitness value in an output file called **GAoutput.txt**. More specifically, each line of the file should contain four items separated by whitespace, in the following order:
(1) the generation number,
(2) the average fitness of the whole population for this generation,
(3) the fitness value of the best strategy of this generation,
(4) the best strategy itself.

You may also want to have your GA periodically demo the best strategy found so far (every 10 or 20 generations, for example). That way, as the evolution proceeds, you can watch Robby's progress as it learns to clean up the environment.

You should also experiment with different GA settings (population size, number of generations, crossover and mutation rates, etc.) to see how quickly your GA can discover a really good strategy.

Save the best strategy your GA found in a text file called **bestStrategy.txt** for submission

In your report:

- Write a 1-2 page summary of your experiments, including the GA parameter settings that produced the best strategy.
- You should discuss how efficient the GA is as well as the variation between experiments.
- Be sure to use evidence (screenshots, graphs, etc) to back your claims.

APPENDIX 1 : Robby World Commands

- **rw.getCurrentPosition()** returns the current 0-based *row*, *column* position of Robby in the grid.
- **rw.getPercept()** returns the percept string specifying what Robby currently sees to the North, South, East, West, and Here.
- **rw.getPerceptCode()** returns the code number of the current percept string as an integer in the range 0-242.
- **rw.distributeCans(*density*=0.50)** randomly distributes soda cans throughout the world, with $0 \leq density \leq 1$ specifying the probability of a can occupying a grid cell. The default value is 0.50.
- **rw.goto(*row*, *column*)** moves Robby to the specified grid location.
- **rw.performAction(*action*)** causes Robby to perform the specified action, where *action* is one of the strings "MoveNorth", "MoveSouth", "MoveEast", "MoveWest", "StayPut", "PickUpCan", or "MoveRandom". The following abbreviations are provided for convenience:
  - **rw.north()** = rw.performAction("MoveNorth")
  - **rw.south()** = rw.performAction("MoveSouth")
  - **rw.east()** = rw.performAction("MoveEast")
  - **rw.west()**= rw.performAction("MoveWest")
  - **rw.stay()**= rw.performAction("StayPut")
  - **rw.grab()**= rw.performAction("PickUpCan")
  - **rw.random()**= rw.performAction("MoveRandom")
  - **rw.look()**= rw.getPercept()
- **rw.graphicsOff(*message*="")** turns off the graphics. This is useful for simulating many actions at high speed when evaluating the fitness of a strategy. The optional *message* string will be displayed while the graphics are turned off.
- **rw.graphicsOn()** turns the graphics back on, and updates the grid to reflect the current state of the world.
- **rw.show()** prints out a non-graphical representation of the current state of the world.
- **rw.demo(*strategy*, *steps*=200, *init*=0.50)** turns on the graphics and demos a strategy for the specified number of simulation steps (default 200) with a randomized soda can density of *init* (default 0.50). Optionally, *init* can be a string specifying the name of a grid world configuration file created with the save command.
- **rw.save(*filename*)** saves the current grid world configuration as a text file, where *filename* is a string.
- **rw.load(*filename*)** loads a grid world configuration from a text file, where *filename* is a string.
- **rw.strategyM** is a string representing the hand-coded strategy *M* created by Melanie Mitchell, as described in the handout.

APPENDIX 2 : Scoring Rubric

The scoring rubric **for your report** is based on the Kentucky General Scoring Rubric from the Kentucky Department of Education (KDE).

Score	**Description**
Category 4 (Score 90%-100%)	● The student completes all important components of the task and communicates ideas clearly. ● The student demonstrates in-depth understanding of the relevant concepts and/or process. ● Where appropriate, the student chooses more efficient and/or sophisticated processes. ● Where appropriate, the student offers insightful interpretations or extensions (generalizations, applications, analogies).
Category 3 (Score 70%-90%)	● The student completes most important components of the task and communicates clearly. ● The student demonstrates an understanding of major concepts even though he/she overlooks or misunderstands some less important ideas or details.
Category 2 (Score 60%-70%)	● The student completes some important components of the task and communicates those clearly. ● The student demonstrates that there are gaps in his/her conceptual understanding.
Category 1 (Score 10%-60%)	● The student shows minimal understanding. ● The student addresses only a small portion of the required task(s).
Category 0 (Score 0)	● Response is totally incorrect or irrelevant.
Blank (Score 0)	● No response.