University of Alberta
Computing Science Department
XAI in Games - W20

Assignment 1
20 Marks
Due date: 03/03 at 23:59

Overview

In this assignment you will implement the evolutionary algorithm we have seen in class for synthesizing scripts for the board game of Can't Stop. It is fine to discuss the assignment with your classmates, but do not share your code with them. We will use the forum on eClass to discuss the assignment. As a reminder, 5% of the marks in this course are from class participation, which also includes being active in the forum, asking and answering questions. The assignment is divided into three parts.

Part I

One of the main goals of this course is to understand algorithms able to derive interpretable and explainable strategies for games. Computer and board games are excellent testbeds for these algorithms because they were designed to be easy to understand and to be challenging and interesting to humans.

In this first part you will learn about the board game Can't Stop by playing a few matches at the Board Game Arena (`https://boardgamearena.com`). You can sign up for free and play against other people or against a computer program. The webpage for the game at the Board Game Arena include the instructions and a few videos showing how to play the game. The webpage also contains a section called "Strategy", which gives a few good hints of how to play the game. The strategy section will help you in the second part of the assignment, where you will manually write a script for playing a smaller version of the game.

Part II

In the second part of the assignment you will write a script for playing a smaller version of the game Can't Stop. Instead of playing on a board with 11 columns, numbered from 2 to 12, the script you will write will play on a board with 5 columns, numbered from 2 to 6. In the original Can't Stop game, the columns from 2 to 12 have 3, 5, 7, 9, 11, 13, 11, 9, 7, 5, 3 steps. In our smaller version of the game, there are 3, 5, 7, 5, 3 steps in the columns from left to right. Instead of using four six-sided dice, as in the original game, we use four three-sided dice. Similarly to the original game, the first player to conquer 3 columns wins the game. The code made available with this assignment supports only two players. An instance of the class Game can be created for the smaller version of Can't Stop as follows.

```
game = Game(n_players = 2, dice_number = 4, dice_value = 3,
column_range = [2, 6], offset = 2, initial_height = 1)
```

You will use a set of functions made available by a domain-specific language (DSL) implemented in the project (see player/scripts/DSL.py) to create your script for the smaller version of the game. Here is an example of a script using a function provided by the DSL (see player/scripts/player_test.py):

```
from players.player import Player
from players.scripts.DSL import DSL
```

```
class PlayerTest(Player):

    def get_action(self, state):
        actions = state.available_moves()

        for a in actions:
            if DSL.isDoubles(a):
                return a
        return actions[0]
```

The file main-xai.py contains the code for testing PlayerTest against a random player. Run main-xai.py and observe the results. Now switch the roles: if PlayerTest was player 1 in the first experiment, then switch it to player 2 and run the code again. Perhaps you will not notice much of a difference in this experiment because the two players are weak. However, you will notice in other experiments that player 1 has an advantage over player 2. This will be important in Part III of the assignment.

Now it is time to study the functions available in the DSL (look for the static methods in the file DSL.py) and write your own script for playing Can't Stop. Try to write the best script possible.

Part III

Implementing the Basic Evolutionary Algorithm (50 Marks)

All scripts generated by your algorithm will have a for-loop iterating over all actions available. This is known as a *sketch*, which is a structure that has to be completed by the synthesizer. The code below shows the sketch you will use, with the curly brackets showing what has to be derived by your algorithm. Your algorithm will also define the number of if-clauses to be included in the script.

```
from players.player import Player
from players.scripts.DSL import DSL

class PlayerTest(Player):

    def get_action(self, state):
        actions = state.available_moves()

        for a in actions:
            if {CODE TO BE DEFINED BY THE SYNTHESIZER}:
                return a
        return actions[0]
```

The evolutionary approach you will implement is the one we discussed in class, which is shown in the pseudocode below. You do not have to implement the evolutionary approach as described in the pseudocode. The pseudocode is here just to give you some guidance of what can be implemented, but feel free to experiment with variations. Here is description of the genetic operations used in the pseudocode:

- **Initialization:** The population is initialized with a set of randomly generated scripts. You can use the context-free grammar in class DSL to generate random scripts. One can start with the initial symbol

```
def EZS(Scripts Z, generations l, pop n, elite e, tournament t, mutation r):
  P = Init(Z, n)
  for _ in range(l):
    Evaluate(P)
    P' = $\emptyset$
    P' = P' $\cup$ Elite(P, e)
    while |P'| < |P|:
      p1, p2 = Tournament(t, P)
      c = Crossover(p1, p2)
      Mutation(c)
      P' = P' $\cup$ c
    P = P'
  Evaluate(P)
  return individual in $P$ with largest fitness
```

of the grammar and randomly select rules from the grammar, the procedure continues while there are non-terminal symbols to be transformed into terminal symbols.

- **Evaluation:** Each script can be evaluated by playing all the other scripts in the population. However, remember that Can't Stop is a stochastic game, thus each script should play every other script in the population a number of times. Moreover, since player 1 has an advantage over player 2, each script should assume both roles during evaluation. In my experiments I noticed that 20–40 matches of each script against every other script (50% of the matches as player 1 and 50% as player 2) produced reliable results. The fitness value of a script is given by the sum of vitories minus the number of losses.

  Some scripts are so weak that they can't even finish a match. If two of such scripts play one another, then the match might never finish. That is why it is a good idea to cap the number of rounds in a match, and if the match reaches that cap, then assign a loss to both players.

- **Elite:** In this operation one selects the best e scripts from the population according to their fitness values. Elites of size 5–10 performed well in populations of size 10–30 in my experiments.

- **Tournament:** The tournament selects a number t of scripts at random from the population and returns the best script from that pool. Tournaments of size 5–10 worked well in populations of size 10–30 in my experiments.

- **Crossover:** This operator will mix the if-clauses of two scripts. In a crossover one chooses a splitting point for the parent script p_1 and all if-clauses before the splitting point are copied to a child script c_1. The procedure is repeated for parent script p_2 and all the if-clauses after the splitting point are copied to c_1, making the child a mixture of p_1 and p_2. Other forms of crossover are possible, feel free to experiment with variations of the one described above.

- **Mutation:** In this operation you will iterate through all if-clauses of a given script and with some probability (0.5 worked well in my experiments) you will replace it by another randomly generated if-clause.[1] With the same probability you will add a random if-clause before or after the current if-clause. Similarly to the other operations, variations are also possible with the mutation operator.

The scripts synthesized by your algorithm will be evaluated by playing the game. This will be done by writing the scripts into .py files and instantiating the scripts during evaluation. In Python this is done as shown in the code below, which dynamically creates an instance of class PlayerTest.

[1]Use the context-free grammar to generate a random if-clause as mentioned in class.

```
import importlib
module = importlib.import_module('players.scripts.player_test')
class_ = getattr(module, 'PlayerTest')
instance_script1 = class_()
```

Python caches the classes instantiated during execution, which can cause trouble to your algorithm. For example, in its first generation the evolutionary approach might synthesize a script named Script1993 and in the next generation another script with the same name is synthesized. Due to its caching scheme, the Script1993 evaluated in the second generation is the one from the previous generation.

There are two solutions to this problem. The first is to never synthesize two scripts with the same name. This way we know there won't be caching conflicts. The other is to erase the folder __pycache__ at the end of each generation. In my implementation I never generate two scripts with the same name and I also erase the folder __pycache__ as it can grow very large with the number of generations.

Removing Unused Rules (20 Marks)

During the evolutionary procedure you will notice the generation of scripts with Boolean expressions that always return false. You will also notice the repetition of if-clauses. These unused if-clauses insert noise into the evolutionary procedure making it difficult to encounter strong scripts.

You will implement a mechanism for removing unused rules from the scripts. That is, after evaluating a population you will remove all rules that were never used in the matches played during evaluation.

Experiments (10 Marks)

You will compare the script synthesized by the evolutionary approach with the best script from the first generation of the procedure. You will also compare the synthesized script with the script you wrote in Part II of the assignment. Recall that both the game and the algorithm are stochastic, so you might have to run the evolutionary algorithm and the matches a few times to perform a fair comparison.

Report (20 Marks)

The report is worth 20 marks. However, if you do not deliver the report you will not earn marks for the earlier parts of the assignment. The report must be submitted through eClass by 03/03 at 23:59. The report, which must be in pdf format, should contain:

- A link to your code in GitHub. Alternatively, you can upload a zip file with your code onto eClass.
- The script you wrote with a description of what you tried to achieve.
- An explanation of how your system removes unused rules from the scripts.
- A detailed presentation of the results. Choose carefully how to present them (e.g., tables and/or plots).
- A discussion of what you have learned about the smaller version of Can't Stop. In this discussion you should compare your script with the script found by your algorithm. Were you surprised by the solution found by your algorithm? Do you think the smaller version of Can't Stop is an interesting game? How could you use the script synthesized by your algorithm to argue in favor or against the commercial development of the smaller version of Can't Stop?