UNIVERSITY OF ALBERTA

University of Alberta
Computing Science Department
Program Synthesis in XAI - W21

Assignment 3
20 Marks
Due date: 29/03 at 23:59

Overview

In this assignment you will implement a system for synthesizing strategies for playing Can't Stop. The assignment is divided into three parts: playing the game, implementing a synthesizer, and writing a report.

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 Part I 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. In Part III of this assignment you will explain the strategy your synthesizer has encountered for the game, thus it is important to understand well how to play the game.

Part II (14 Marks)

In the second part of the assignment you will implement Bottom-Up Search (BUS) with Iterated Best Response (we referred to it as "self-play" in our lectures).

Game of Can't Stop

The code starter provided in this assignment implements the game of Can't Stop and the domain-specific language you will use. The game of Can't Stop can be instantiated as follows.

```
game = Game(n_players = 2, dice_number = 4, dice_value = 6, column_range = [2,12],
            offset = 2, initial_height = 3)
```

The parameters allow us to instantiate other versions of the game. We will use the traditional version of the game, with columns ranging from 2 to 12, identical to the website Board Game Arena. See file `main-starter.py` for an example of how to run matches between a random agent (chooses actions randomly) and a manually constructed programmatic strategy that uses our DSL. You will notice that each agent plays half of the matches as player 1 and the other half as player 2. This is to ensure fairness with the two strategies as player 1 has a small advantage in Can't Stop. You should follow the same approach in your experiments.

Domain-Specific Language

The starter code provides the implementation of the following DSL, where I is the initial symbol.

$$
\begin{aligned}
I &\rightarrow \text{ sum}(L) \mid \text{ argmax}(L) \\
L &\rightarrow \text{ map}(\text{lambda}(E), L) \mid \text{ neutrals } \mid \text{ actions} \\
E &\rightarrow I \mid \text{ map}(\text{lambda}(E), L) | S | S \oplus S \\
S &\rightarrow \text{NumberAdvancedThisRound} \mid \text{NumberAdvancedByAction} \mid \text{IsNewNeutral} \mid A \\
A &\rightarrow \text{progress\_value}[\text{column}] \mid \text{move\_value}[\text{column}]
\end{aligned}
$$

Functions sum and argmax are the implementations given by Numpy and map and lambda are Python's regular implementations (see file `DSL.py` for details). The implementation of a lambda function can contain any instruction used in the language, as it considers non-terminal symbols I and E.

The non-terminal symbol S provides three domain-specific functions.

- `NumberAdvancedThisRound` returns the number steps performed in a given column for the current round (i.e., we have placed a neutral marker, but since the player hasn't stopped playing, the neutral markers haven't been converted into the player's color).
- `NumberAdvancedByAction` returns the number of steps an action performs, which can be either 1 or 2.
- `IsNewNeutral` returns True if the action uses a new neutral piece or False, otherwise.

The operator $\oplus$ represents the basic operations: $+, -, *$. The non-terminal symbol A allows the synthesized strategies access to two arrays with domain-specific information. The arrays are accessed with a local variable `column` from a lambda function. See `interpret_local_variables` in the file `DSL.py` for details. A similar scheme is implemented for the parameters of the three functions described above.

Sketch

The DSL described above will be used to synthesize two instructions in the function `get_action` of the sketch below, which was extracted from file `rule_28_sketch.py`.

```
def get_action(self, state):
    self._state = state

    self.actions = self._state.available_moves()

    neutrals = [col[0] for col in self._state.neutral_positions]

    if self.actions == ['y', 'n']:
        if self.will_player_win_after_n():
            return 'n'

        elif self.are_there_available_columns_to_play():
            return 'y'
        else:
            env = self.init_env()
            score = self.program_yes_no.interpret(env)
            if score >= self.threshold:
                return 'n'
            else:
                return 'y'
    else:
        env = self.init_env()
        return self.actions[self.program_decide_column.interpret(env)]
```

One needs to make two types of decisions in the game of Can't Stop: whether the continue playing or not (yes-or-no decision) and which columns to play, given a set of possibilities (column decision). You will use Bottom-Up Search to synthesize the program for dealing with the column decision (variable `self.program_decide_column`). You should use the program provided as example in `main-starter.py` as the variable `self.program_yes_no` while synthesizing `self.program_decide_column`.

3 Extra Marks. Implement a synthesizer that produces programs for both the yes-no and the column decisions simultaneously. You will receive the extra marks if your synthesized program wins at least 48% of 2,000 matches against the programmatic strategy provided as example in `main-starter.py`.

Evaluation Function

In previous assignments we had a set of input-output training pairs that allowed us to verify whether the synthesis problem was solved. In this assignment we will use Iterated Best-Response (IBR) instead. We start with a "reasonable" strategy σ_0 and we will search for an approximated best response σ_1 for the strategy σ_0. Then, in the next iteration we search for an approximated best response for σ_1. This process is repeated a number of times. After we have determined σ_0, we will perform 5 iterations of the IBR algorithm.

We will use BUS to find σ_0 by searching for a program that is able to finish (win or lose) 60% of 100 matches played against itself. Then, we will start the iterations of IBR, where a strategy σ_{i+1} is considered an approximated best response to strategy σ_i if the former defeats the latter in at least 55% of the matches played. Ideally, σ_i and σ_{i+1} will play 1,000 matches to determine if we have encountered an approximated best response. However, it is too expensive to play 1,000 matches for every program encountered in search. Instead, we will use a **triage-based evaluation function**, where strategy σ plays 10 matches against σ_i (the strategy we are approximating a best response to) and, if σ wins at least 2 matches, we play another 190 matches. If σ defeats σ_i in at least 55% of the total 200 matches, then they play another 800 matches. We consider σ an approximated best response of σ_i if it wins at least 550 matches of the 1,000 matches played.

The idea behind this triage-based scheme is to spend little time evaluating weak strategies and spend more time evaluating more promising strategies. We you will also only evaluate programs of type 'Argmax' for the column-decision program. This is because it is too expensive to evaluate all programs generated in search.

Searching with IBR

You can assume that the approximated best responses for strategy σ_i are in programs expanded after σ_i in the BUS search. That way, once you find a best response σ_i to σ_{i-1}, you will continue searching for a best response for σ_i from where you left off (no need to restart the search from scratch). Note that this isn't true in general and a best response σ_i to a strategy σ_{i-1} might have fewer AST nodes than σ_{i-1}.

Your synthesizer will use a signature similar to the following.

```
br, num = bus.synthesize(10,
            [Sum, Map, Argmax, Function, Plus, Times, Minus],
            ['neutrals', 'actions'],
            ['progress_value', 'move_value'],
            [NumberAdvancedThisRound, NumberAdvancedByAction, IsNewNeutral],
            eval,
            programs_not_to_eval)
```

Here, `br` is the best response to the current strategy, which is stored in object `eval`. The object `eval` encodes the evaluation function, which can be either triage-based, non-triage based, or simply encode a function that verifies if a program is able to finish the game in self play (used in defining σ_0). The number 10 specifies the maximum search size and the lists provide the DSL features used in search: operators, lists, scalars obtained from lists, and functions, respectively. Finally, `programs_not_to_eval` stores information of the programs evaluated in the previous iteration of IBR, which allows the search to continue from where it left off.

Part III (6 Marks)

The report is worth 6 marks, but if you do not deliver the report you will not earn marks for the earlier parts of the assignment. The writeup, which must be submitted as a pdf file, should

- report the running time of your algorithm as well as the number of programs generated and evaluated (recall that you will only evaluate programs of type 'Argmax', but will generate many more programs).
- report the running time of your algorithm without the triage-based evaluation (i.e, evaluate every program with 1,000 matches of the game). Discuss the difference in running time of the triage with the non-triage versions of your system.
- describe the strategy synthesized by your method. What is this strategy doing?
- explain how you could improve your results in terms of the strength of the strategy synthesized.

In addition to the items above, please also include details of your implementation that are needed to understand your results. For example, you might have implemented an enhancement that isn't described in the assignment but that played an important role in your experiments.