COGS 125 / CSE 175
Introduction to Artificial Intelligence
Specification for Programming Assignment #3

David C. Noelle

Due: 11:59 P.M. on Friday, December 1, 2023

Overview

This programming assignment has three main learning goals. First, the assignment will provide you with an opportunity to practice developing programs in Python using the PyCharm integrated development environment. Second, this assignment will allow you to practice implementing the decision theory principle of maximum expected utility in a game-playing context. Your work on this assignment is intended to strengthen your understanding of how expected utility values are calculated, including how they are calculated in multi-agent environments. Finally, this assignment requires you to design a heuristic evaluation function for a game, providing you with experience in the design of such functions.

You will be provided with Python source code that (almost) implements a two-player game called the *Guardian Game*, with a single computer player pitted against a single human player. You are charged with completing the implementation of the computer player's decision procedure by writing a function that calculates an expected utility value and another that computes a heuristic evaluation value for a state of play.

Submission for Evaluation

To complete this assignment, you must generate one Python script file, called "`heuristic.py`". That file is to contain two functions: a component of a *minimax* look-ahead search procedure called `expected_value_over_delays` and a heuristic evaluation function for non-terminal states of game play called `heuristic_value`. This is the only file that you should submit for evaluation.

To submit your completed assignment for evaluation, log onto the class site on CatCourses and navigate to the "Assignments" section. Then, locate "Programming Assignment #3" and select the option to submit your solution for this assignment. Provide your single program file as an attachement. Do not upload any other files as part of this submission. Comments to the teaching team should appear as header comments in your Python source code file.

Submissions must arrive by 11:59 P.M. on Friday, December 1st. Please allow time for potential system slowness immediately prior to this deadline. You may submit assignment solutions multiple times, and only the most recently submitted version will be evaluated. As discussed in the course syllabus, late assignments will not be evaluated and will receive *no credit*.

If your last submission for this assignment arrives by 11:59 P.M. on Tuesday, November 28th, you will receive a 10% bonus to the score that you receive for this assignment. This bonus is intended to encourage you to try to complete this assignment early.

Activities

The Guardian Game

The *Guardian Game* is a two-player fully-observable zero-sum game that includes some randomness and, therefore, some uncertainty about the outcomes of actions. The board represents a single walkway extending to the West and to the East of a magical statue called the *Guardian*. One player begins the game along the walkway to the West of the Guardian and the other begins along the walkway to the East. They begin at equal distances from the Guardian. The players take turns sneaking toward the Guardian, with the goal of reaching the Guardian before the other player does. The walkway is divided into "squares" or "steps", and each turn involves a player selecting a number of steps to move on that turn (i.e., a move *action*). Once the length of the move is selected, the player begins to move toward the Guardian, but there is danger in approaching the magical statue. The Guarding can turn its head, facing either West or East. If a player is moving when the Guardian is facing toward them, a magical ray of light springs from the Guardian's eyes, banishing that player to another world, causing them to lose the game. This means that each player's turn begins with the Guardian facing in the other direction. Each move action should be selected so as to complete the move before the Guardian turns its head to look in the direction of the player. If the move is completed before the head rotates, the other player is allowed to take a turn. If the head faces the player before the move is done, catching the player in motion, that player loses, and the other player wins the game. If neither player is caught in motion, turn after turn, then the first player to reach the Guardian wins.

The Guardian turns its head after a random number of time steps. This interval is always at least 2 time steps, so a player is safe on any turn that they select the action of moving only 1 step. If the player opts to move 2 or more steps, however, there is a chance that they will be caught in motion by the Guardian. There is a maximum length to this interval, as well, so any move action of that length or longer is guaranteed to result in the player being caught. Thus, on each turn, the current player selects an action ranging from 1 step to one less than the length of the maximum delay before the Guardian turns its head.

This game has a number of parameters that specify details of game play, with these parameters specified as variables or functions in the Python implementation of the game. These things include:

- The starting distance to the Guardian is a number of steps specified by `board_size`, which defaults to 13 steps.

- The computer always is the West player, and the human opponent is always the East player. The player that gets the first turn is determined by a (digital) coin toss at the beginning of the game.
- The smallest action size is specified by `min_act_steps`, which defaults to 1. The largest action size is specified by `max_act_steps`, which defaults to 4. Actions must be an integer number of steps between these bounds, inclusive.
- By default, the delay prior to the Guardian turning its head is a random value between `min_time_steps` and `max_time_steps`, defaulting to 2 and 5, respectively. This value is determined by (digitally) flipping three fair coins and adding two to the total number of "heads" that result. Note that this is not a uniform distribution, as the random delay values vary in their probability of occurrence.
- The game ends when either a player is banished due to being caught in motion or when a player reaches the Guardian. The terminal state of the game is assigned a *payoff*, with a positive value of `max_payoff` if the computer player wins and a negative value of $(-1 \times$ `max_payoff`$)$ if the human opponent wins.

Because of the use of randomness, there is no strategy that guarantees a win, but some action choices are better than others. The goal of this assignment is to complete the provided Python program so as to make it as competitive as possible.

Implementation

The code for a Python implementation of the minimax approach to selecting actions is contained in a collection of provided script files. These include:

- "`parameters.py`" — This file contains a small set of game parameters, including the board size, the range of allowable actions, and the maximum payoff.
- "`game.py`" — This file provides the `Game` class. This class implements both a game session and the current state of play. For example, it records where the players are currently located on the walkway, whose turn it is, what action the current player has selected, and how many time steps remain before the Guardian will turn its head. The class includes a method called "`play`" which implements a session of play, from the initialization of the board to declaring a winner.
- "`main.py`" — This is a top-level driver script, performing needed initialization, creating a `Game` object, and calling the `play` method.
- "`minimax.py`" — This file provides a collection of functions that, together, produce a minimax look-ahead search implementation, allowing the computer player to select actions. The entry point to this code is the function called "`minimax_action`", which examines each possible action choice for the computer player and selects the action that maximizes the value of the resulting state. The value of a state of play is returned by the "`value`" function.

The action selection process combines two important artificial intelligence concepts.

First, it incorporates minimax search. The value of a considered future state of play, where one of the players is about to select an action, is determined by recursively computing the values of the states that result from each possible action, and selecting the action corresponding to the minimax value. If the current player of the given state is the computer player, then the action that *maximizes* the value determines the backed-up value of the state. If the current player is the human opponent, then the action that *minimizes* the value determines the backed-up value. This is the standard minimax approach, with the computer player seeking high value states and the human opponent seeking low value states.

Second, the implementation makes use of the concept of *maximum expected utility*. The randomly selected delay prior to the Guardian turning its head produces resulting states of game play that vary in their values. Succinctly, low random delays are bad for the player currently taking a turn, as they result in banishment for a broader range of move actions. In order to deal with this uncertainty, the value of a game state just prior to randomly selecting the delay is taken to be the *expected utility* of that state – the *expected value* over the states that result. Recall that this is the sum, over all possible delays, of the values of the resulting states (as returned by the `value` function), with each value weighted by the probability of randomly selecting the corresponding delay. (Note that a function called "`probability_of_time`" is provided, which calculates the probability of a given delay.) Combining this expected value calculation with minimax search provides a way to address the randomness in the game.

As is true in many games, it would take too much time to perform a look-ahead search all the way to terminal states. Instead, the provided implementation looks ahead to a ply specified in "`parameters.py`" as `max_ply`. At that point, non-terminal states are assigned values using a heuristic evaluation function. This heuristic evaluation function must be fast, avoiding all further look-ahead search, but it should provide an estimate of the future payoff attainable from the given state. A positive value for the heuristic indicates that the state of play is looking good for the computer player, while a negative value indicates that the human opponent is ahead. The heuristic function must be *symmetric*, in the sense that if the two players swapped positions, the heuristic value would become the additive inverse of the original value. For example, if a given state of play has a heuristic value of 24, then another state of play that is identical to this one, except for the fact that the two players have swapped positions, should have a heuristic value of -24. Heuristic values must also be within the range of possible payoff values for terminal states.

Note that look-ahead search can be quite slow, especially as the maximum ply grows. Thus, a good heuristic function is important for timely decisions by the computer player.

Activities

For this assignment, you are to complete the look-ahead search implementation by providing two functions. Both of these functions are to be written in a file called "`heuristic.py`", and these functions must work with the other provided Python script files *without modifying the other files in any way*. All of these files, including a template for "`heuristic.py`", are available in a ZIP archive file called "`PA3.zip`" in the "Assignments" section of the class CatCourses site, under "Programming Assignment #3". The contents of these Python code files will be briefly discussed

during a laboratory session, and comments in these files should assist in your understanding of the provided code. Questions are welcome, however, and they should be directed to the teaching team.

The first of the two functions to be implemented is `expected_value_over_delays`. This function takes a `Game` object as input, representing a state of play in which the current player has chosen an action, but the time delay before the Guardian turns its head has not yet been randomly determined. This function is to return the *expected value* of the state over all possible random delays.

The second function to be written is `heuristic_value`. Without performing any additional search, this function must quickly estimate the value of the given state of play, encoded by a `Game` object. The task of designing this function is an opportunity for creativity. There are many ways to assign heuristic values to game states, and the quality of these estimates can greatly affect the performance of the computer player.

Both of these two functions are called by the `value` function in "`minimax.py`", and they are needed for the minimax procedure to work correctly. Well written functions will result in competitive play from the program. Once again, both functions are to written in the "`heuristic.py`" file, which is to be submitted for evaluation.

Evaluation

Your submission will be evaluated primarily for accuracy, with efficiency being a secondary consideration. Your source code *will* be examined, however, and the readability and style of your implementation will have a substantial influence on how your assignment is evaluated. As a rough rule of thumb, consider the use of good software writing practices as accounting for approximately 10% to 20% of the value of this exercise. Please use the coding practices exemplified in the provided utility files as a guide to appropriate readability and style. Note also that, as discussed in the course syllabus, submissions that fail to run without crashing on the laboratory PyCharm IDE will not be evaluated and will receive *no credit*.

The functions that you implement should write *no output*, as this will clutter the output produced by the "`main.py`" script. If you include any statements that write output in your code (perhaps as tools for debugging) these should be removed prior to submitting your code files for evaluation. You may receive *no credit* for your submitted solution if it produces extraneous output.

As for all assignments in this class, submitted solutions should reflect the understanding and effort of the individual student making the submission. **Not a single line of computer code should be shared between course participants. Not a single line of computer code should be received from any other person, such as students not enrolled in the class and people providing tutorial assistance.** If there is ever any doubt concerning the propriety of a given interaction, it is the student's responsibility to approach the instructor and clarify the situation *prior* to the submission of work results. Also, helpful conversations with fellow students, or any other person (including members of the teaching team), should be explicitly mentioned in submitted assignments (e.g., in comments in the submitted source code files). These comments should also explicitly mention any written resources, including online resources, that were used to complete the exercise. Citations should clearly identify the source of any help received (e.g., "Dr. David Noelle" instead of "the professor", "The Python Tutorial at `docs.python.org/3/tutorial/`" instead of "Python

documentation"). Failure to appropriately cite sources is called *plagiarism*, and it will not be tolerated! Policy specifies that detected acts of academic dishonesty must result minimally with a zero score on the assignment, and it may result in a failing grade in the class. Please see the course syllabus for details.

To be clear, please note that all of the following conditions must hold for a submitted solution to receive *any* credit:

- solution submitted by the due date
- solution runs without crashing on the laboratory PyCharm IDE
- solution produces no extraneous output
- solution works with unmodified utility code, as provided
- solution does *not* include code written by another person (with or without modifications)
- solution explicitly cites all help received, whether from a person or some other source

While partial credit will be given for submissions that meet these conditions, failure to meet one or more of these conditions will result in *no credit* and, in the case of academic dishonesty, may result in a failing grade in the course. Once again, please see the course syllabus for details.

The members of the teaching team stand ready to help you with the learning process embodied by this assignment. Please do not hesitate to request their assistance.