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

David C. Noelle

Due: 5:00 P.M. on Monday, December 9, 2013

Overview

This programming assignment has three main learning goals. First, the assignment will provide you with an opportunity to practice your skills in developing programs in Java™ using the Eclipse 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 Java™ source code that implements a two-player version of the Zombie Dice™ game, with a single computer player pitted against a single human player. This provided code will be fully functional, but the computer player performs very poorly in this version, due to missing components in its move selection algorithm. You will be required to complete 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 modify only a single Java™ class source file, called "`Eval.java`". A template for this file will be provided to you, along with additional supporting source code. This file contains a collection of static functions that may be used to assess the "value" of states of play, estimating the degree to which that state of play is likely to lead to a win for the computer player or the human user. As provided, this source code file is *not* complete, however. It is missing a full implementation of a static function called `value_roll_hand` and another called `heuristic`. Your task is to provide bodies for these two static functions by modifying the "`Eval.java`" file. You should *not* modify this file except to provide these two static function

implementations. Your implementations should work well with the rest of the provided source code, without any other modifications to that code. Thus, your modified version of "`Eval.java`" is the only file that you should submit for evaluation.

To submit your completed assignment for evaluation, log onto the "`F13-INTRO TO AI`" site on `UCMCROPS` and navigate to the "Assignments" section. Then, locate "Programming Assignment #3" and select the option to submit your solution for this assignment. Provide your program file as an attachment. Comments to the teaching team should largely appear as comments in your Java™ source code file, but you may also include brief written comments using the assignment submission web form.

Submissions must arrive by 5:00 P.M. on Monday, December 9th. 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.

If your last submission for this assignment arrives by 5:00 P.M. on Thursday, December 5th, 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

You are to provide two Java™ functions, implemented as static methods of the provided `Eval` class. Your provided Java™ source code must be compatible with a collection of provided Java™ classes that implement a two-player version of the Zombie Dice™ game. Indeed, your assignment solution will be evaluated by combining your submitted modified `Eval` class file with copies of the provided utility files and testing the resulting complete program against a variety of test cases. In other words, *your solution must work with the provided utilities, without any modifications to these provided files*.

More specifically, you are to provide the following two static functions, all appearing in the "`Eval.java`" source code file:

- `static double value_roll_hand (State s, int depth)`

  This static function returns a real-valued evaluation of the quality of the given state of play, `s`. In general, this value is positive if the state is good for the computer player and negative if the state is good for the human user. This function assumes that the given state is one in which the player whose turn it is has already drawn a handful of dice from the dice cup and is about to roll them. Thus, the function must calculate the expected utility of the state by considering every possible outcome of the dice roll and the resulting utility value of each of those possible outcomes. The utility values of the outcomes are to be calculated using the *minimax* algorithm for game-tree search, looking ahead to recursively calculate the expected utility values of the potential future outcome states. You need not implement this look-ahead process, however, as a provided function called `value_rolled_hand` already calculates the expected utility of a game state in which the dice have already been rolled. The `depth`

argument to this function, recording the depth of the look-ahead search, need only be passed on to any calls to `value_rolled_hand`.

- `static public double heuristic (State s)`

  This static function returns a real-valued evaluation of the quality of the given state of play, `s`. This function should *not* perform any game-tree search in order to determine this value, however. Instead, it should quickly calculate a heuristic estimate of the quality of the current state of play. This function is used to evaluate non-terminal states at the maximum depth of the game tree. As before, the evaluation value should be positive if the state is good for the computer player and negative if the state is good for the human user. The function should handle any valid state of play.

Note that a member function of the provided `State` class, called `payoff`, assigns a win for the computer player a value of `State.win_payoff` and a win for the human user a negative value of equal magnitude. Thus, all expected utility values, including that returned by the `heuristic` function, should be bounded between plus and minus `State.win_payoff`.

If these two static functions are properly implemented, the `main` method in the provided `Pthree` class should play a strong game of Zombie Dice™. In order to properly implement these functions, it will be important to understand how the game of Zombie Dice™ is played. In this game, each player potrays a zombie, hungry to eat the brains of human victims. The winner of the game is the player who successfully eats the most brains. This game is produced by Steve Jackson Games, and information about it may be found at:

`http://zombiedice.sjgames.com/`

This game is very easy to learn. You will be provided with the one page rules document for this game as a PDF file.

The Java™ utility classes that you are required to use are provided in a ZIP archive file called "`PA3.zip`" which is available in the "Resources" section of the class `UCMCROPS` site. These utilities include:

- `Die` — This object encodes one of the thirteen dice used in the Zombie Dice™ game. Each die represents a potential victim of a zombie attack. Green dice represent easy to catch victims. Red dice represent hard to catch victims. Yellow dice are of moderate difficulty. Each die has six faces, with each face bearing one of three outcome symbols. A "brain" face indicates that the zombie player has captured this victim's brain. A "blast" face (or "shotgun") indicates that the zombie player has been shot by this potential victim. A "feet" face indicates that this potential victim has run away.

- `Cup` — The Zombie Dice™ game includes a dice cup which initially contains all thirteen dice. On each turn, players draw three dice randomly from this cup before rolling them. This object encodes the dice cup, providing tools for drawing dice and tracking the probability that dice of various colors will be drawn.

- `State` — This object encodes a state of play. This may be the current state of play in an ongoing game, or it may be a hypothetical future state of play being considered during game-tree search. Tools are provided for modifying the state of play and for calculating the probability of various successor states.

- `Eval` — This class provides a home for a collection of static functions that implement *minimax* game-tree search to a fixed depth, using a heuristic evaluation function to assess non-terminal states at the maximum depth. The public function called `value` returns the expected utility of an argument `State` object. You are required to implement a portion of this game-tree search algorithm, including the heuristic evaluation function.

- `Game` — This object encodes an actual game session of a two-player version of Zombie Dice™. A function is provided that chooses a computer player move based on the principle of maximum expected utility, with expected utility values calculated by the static functions in the `Eval` class. Also provided is a function for soliciting moves from the human user. The public function called `play` coordinates game play, including displaying each move to the screen.

- `Pthree` — This tiny class simply instantiates an object of the `Game` class and calls its `play` member function, allowing a game of Zombie Dice™ to be played. Your code must allow `Pthree` to offer a challenging gaming experience for the human user.

The contents of these Java™ utility files will be discussed during a course laboratory session, and inline comments in these files should assist in your understanding of the provided code. Questions are welcome, however, and should be directed to the teaching team.

Your implementation 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. Note also that, as discussed in the course syllabus, submissions that fail to successfully compile under the laboratory Eclipse Java™ IDE (i.e., they produce "build errors") will not be evaluated and will receive *no credit*.

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. 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). Failure to appropriately cite sources is called *plagiarism*, and it will not be tolerated!

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.