Kathryn Hodge

CMPU 250

April 23, 2015

Monte Carlo AI in Tic Tac Toe | Program 2

By definition, the Monte Carlo method is “a technique in which a large quantity of randomly generated numbers are studied using a probabilistic model to find an approximate solution to a numerical problem that would be difficult to solve by other methods.” In other words, to find a solution, the Monte Carlo method runs a bunch of simulations, trying to solve the problem, and returns the answer that, in theory, worked the best. The goal of my project was to create a Monte-Carlo based AI for a common game, Tic Tac Toe. In order to do this, I would first have to create a program that implemented the rules of Tic Tac Toe, and then add in the AI later.

The rules of Tic Tac Toe are:

1. X moves first
2. A piece must be placed on any empty space.
3. The user and computer must alternate turns
4. A player wins by being the first to connect a line of friendly pieces from one side or corner of the board to the other.
5. The game ends when either one player wins or it is no longer possible for a player to win (in which case the result is a draw).

For my game to be implemented successfully, it must follow these rules. As for the AI, it must make its turn using the Monte Carlo Method. What exactly does that mean? It means that the AI must run through thousands of simulations of possible first

moves and their repercussions, and after running through these simulations, choose the move that is most likely to make it win.

In my program, Tic Tac Toe game logic is located in the *TicTacToe.java* file and the code managing the AI is in the *MonteCarloAI.java* file. The *TicTacToeApplication* file sets up the user interface and contains all the necessary information for the program to be run. To prove Tic Tac Toe is implemented correctly, I shall dive into *TicTacToe.java*. In the beginning, the board has every spot available, as shown in setBoard() [Line 66], and there is no winner [63]. The user, who is given the character 'X' [52], automatically plays first, via *TicTacToeApplication.java* [38-52]. If the user inputs any spot that is taken, then the user, within *TicTacToeApplication.java,* is sent back to the do clause in the do-while loop and must input another row/column spot combination [53]. Thus, the user must play its token in an empty space. If the user tries to put in a row/column that is not in the bounds of tic tac toe, stringToIntCheck() in *TicTacToeApplication.java* requires the user to input a different number that is within the bounds [85]. The method stringToIntCheck() calls withinRange() from *TicTacToe.java* file and that's how the program knows the boundaries of the board [87]. In the *MonteCarloAI.java* file, the chooseRandomSpot() method can only choose spot candidates that are within the bounds and not taken due to the nature of the for loops [64-65] and the extra check to see if the spot is available [66]. Thus, the game will not put a token in any spot that is not within the bounds of the board or that is already taken.

In *TicTacToeApplication.java*, the user and the AI are forced to switch turns back and forth because of their organization in the while loop [19-81]. The while loop ends once there are no more spots available or one player wins [29 & 63]. This check for a

winner happens not only after an execution of the while loop, but also in between turns, so if the user (or AI) wins, the game stops. To check if someone has won, the isThereAWinner() method is called within *TicTacToe.java* [148]. This algorithm checks all the possible combinations in where a player could win [150-152], and if a player has won, then it sets the winner appropriately [154, 156, 158]. If someone has won, the function returns true; if not, it returns false. This method is called within gameOver() [209], and if there is a winner when this method is called, it will return a string that states who the winner is along with some UI friendly information [212]. This string will be taken and printed out to the user within *TicTacToeApplication.java* and the game will end [63-67 or 29 & 81]. To check for a draw, the program uses gameOver() in *TicTacToe.java* to check that all the slots on the board are filled and there is no winner [213]. The function gameOver uses a helper isTheBoardFilled() [198] to determine if the board is filled. In isTheBoardFilled(), each spot is checked and if they all have some token, either AI or user, then the board is filled and this method returns true [199-207]. However, if there are spots still free on the board, then this method returns false [201-202], making the gameOver() method drop into the last else clause, which returns the string “notOver” [216-218]. Thus, in all scenarios, the program detects a winner or draw if there is one and ends the game accordingly.

Now that the game is proven to work correctly, I must prove that the AI makes its decisions using the Monte Carlo method. In other words, it must run a series of simulations of gameplay based on the current board, choosing random moves along the way, and decide which first move is most likely to produce a win. The key algorithm to this program is runRandomSimulations() [139] that lies in *MonteCarloAI*. This method

keeps track of the “goodness” of each spot using an array called winPoints. Each spot on the board has an entry within winPoints [140] and after every simulation, AI’s first move’s entry is updated based on the results of the game [189-197]. At the beginning of the algorithm, runRandomSimulations() fills each entry in winPoints that corresponds to available spot with 0 [151]. If a spot is not available, the function puts an extremely low value in the corresponding spot’s winPoints entry so that it never picked by the AI as a “good spot” [154]. If every spot is taken on the board, then an exception is thrown because the game should have ended before the AI took its turn [160-161]. After setting up winPoints, the game goes through many simulations to determine the best spot. Keeping track of the first move [167, 182-184] , the AI chooses a random spot on the board [175] and then alternates turns with the user [178], who also chooses random spots. Since the while condition is checked each turn, the game will end when either someone wins or there are no more spots available [172]. Each play is played on a tempBoard, which is copied from the real board, so that the real board from the game is not changed. To allow multiple stimulations to use the same board, the tempBoard is reset to the initial conditions of the inputted game at the end of each round [199]. As mentioned before, the outcome of the stimulated game determines the value of the firstMove’s entry within winPoints [189-197].

After all the simulations have played out, the game is reset for game play [202]. The program has to reset the game because the gameOver() check as well as its helper methods change the variable winner within the game so we must reset the game’s winner back to ‘-‘. We know that the game’s winner variable must have ‘-‘ before because otherwise the real game would have been over before

runRandomSimulations() was called. Next, the algorithm finds the maximum value of all the entries within winPoints [208-213]. In case multiple spots have the same value, the algorithm adds all the “good spots” to an array called goodSpots [219]. Then, it randomly picks one of the good spots within the goodSpot array and returns that spot [227]. This good spot is carried through pickSpot [56] and is played in real gameplay via *TicTacToeApplication* on lines 73-76. Thus, by making random moves, running a series of simulations, and making a final choice by observing the results of these simulations, the MonteCarloAI.java correctly implements an AI based on the Monte Carlo method.

I learned quite a lot from implementing this program. First, the AI does not rely on strategy at all. It has no knowledge of how to win the game, but rather it just knows what possible choices are out there and what moves are available, and keeps track of which moves are proven to win the most. Therefore, instead of trying to make the user lose or get a draw if there is no possible way for the it to win, the AI focuses solely on the positions that are observed to win the most from its stimulations. In addition, the stimulations also assume that the user is making random decisions with no strategy involved, like the AI. Consequently, if the user is one move away from winning, the AI will not know this and will only focus on which moves appear to give it the best chance to succeed. In addition, this AI only focuses on whether a firstMove resulted in a win, loss, or draw. It does not take into account how many moves it took for the AI to get to that win. This means that the AI might be one move away from winning, but other spot choices might allow the AI to win as well, even if they require multiple turns. This could cause the AI to pick a seemingly lesser spot because an obvious choice, the spot that makes an immediate win, might be overshadowed by a different spot that could allow

the AI to win several turns later. However, because there are more chances to lose if a firstMove takes more turns to let the AI win, the algorithm still works because in theory, it permeates all possibilities and winPoints holds a value that totals all the possibilities of a loss or win for each firstMove spot.

In the future, I would want to try implementing a Monte Carlo based AI with another game that relied less on strategy so that the AI would be more difficult. In addition, in my implementation, I use two different representations of spots on the board. There are advantages to both, but in the future, I would want to keep a consistency by just choosing one so that I wouldn't have to convert back and forth. However, for this program, I focused more on the AI aspect rather than the game so I feel like it isn't a huge deal. Furthermore, I would also want to attempt to break up the code in my runRandomSimulations() method. It's almost 100 lines, but I couldn't seem to find a good way to break it up.

In the console folder, there are print outs of actual game plays of the user winning, the AI winning, and a draw. In addition, for each game, there is a corresponding log that reveals the probabilities for each spot for every AI turn, which uncovers the reasoning behind the AI's choice.