Tic Tac Toe +

Final Project

CS152: Harnessing Artificial Intelligence Algorithms
Minerva Schools at KGI

Austin Perzben / Agustín Pérez del Castillo
April 2021

Final Project: Tic Tac Toe +

Problem Definition

The classic two-player game of Tic Tac Toe is a great medium to exemplify the ways in which Artificially Intelligent agents can perform problem-space search in what are sometimes unexpectedly efficient and effective ways. The problem of extending Tic Tac Toe to be played on a 4x4 grid rather than the traditional 3x3 adds layers to the complexity of the searhc task presented to any AI systems attempting to play the game. The minimax algorithm provides a method for conducting this search algorithmically, and improvements can be made upon it through the use of alpha-beta pruning, an extension process that allows an intelligent agent to save time and effort when searching for solutions to the problem by ignoring possible cases that would not affect the outcome of the game, under the assumption that the players behave rationally and play optimally. Furthermore, the understanding that "the value of a position to player A in such a game [zero-sum game, as Tic Tac Toe] is the negation of the value to player B" gives way to the negamax algorithm – an improved version of minimax that reduces the time it takes for an program to conduct the search and determine optimal play strategies. Finally, the use of evaluation functions allow the minimax search to be depth-limited by evaluating the value of a game state. All of these beneficial possibilities for addressing the problem using AI are what informed the majority of my work on this project.

Solution Specification

In order to address the problem of how to determine optimal play strategies in a 4x4 Tic Tac Toe AI player, I worked to implement an interactive command-line interface for the game, an AI player agent that leverages the advantages of the minimax search algorithm, and extended its capabilities by adding alpha-beta pruning to the algorithm in order to decrease the number of explored game states, as well as the improvements introduced by the negamax extension of minimax, which uses the understanding that `max(a,b)=-min(-a,-b)` to allow the AI agent to calculate the best choice for the player regardless of whether it is the maximizing or minimizing payer in the game. Finally, I implemented two different evaluation functions that the agent can utilize when playing against a human opponent and calculating the payoff for a given state of the game.

Analysis of Solution

I tested my AI player using both of the evaluation functions that I implemented, as well as varying maximum depth levels for the improved minimax search algorithm. The agent was able to play optimally using `EvalFunc1()` so long as the maximum depth of search was greater than 4 levels. When using `EvalFunc2()` the agent processed moves and made decisions somewhat slower, but was able to play optimally with a maximum depth of 3 state-search levels only, which led me to the overall conclusion that the second evaluation function provides more accurate representations of the potential payoffs for any given game state. Further work on this project would explore the use of more evaluation functions and potentially even alternatives to the minimax search algorithm. It could be interesting and insightful to try using the same evaluation functions with less-optimized versions of the minimax general algorithmic pattern as well, since this would provide insight into the efficiency-related benefits of using alpha-beta pruning and the negamax extension. Finally, one could attempt to implement a transition table in a prolog knowledge base and give the AI agent access to this KB for it to further optimize it's play strategies.

References

http://zulko.github.io/easyAI/

https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#Improvements_over_naive_minimax

https://ai.dmi.unibas.ch/_files/teaching/fs16/ai/slides/ai42.pdf

https://www3.ntu.edu.sg/home/ehchua/programming/java/javagame_tictactoe_ai.html

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

Appendix: Code

ttt_plus.py

```
#!/usr/bin/python
#
# 4-in-a-row version of Tic Tac Toe two-player game
# author: Austin Perzben
# created: 2021-04-30

from copy import deepcopy
from players import RealPlayer, AIPlayer
from evaluation_functions import EvalFunc1, EvalFunc2


class TTTPlus:
    """ The board positions are numbered as follows:
             1  2  3  4
             5  6  7  8
             9 10 11 12
            13 14 15 16
    """

    def __init__(self, players):
        self.players = players
        self.board = [0 for i in range(16)]
        self.nplayer = 1  # player 1 starts.
```

```
def play(self, nmoves=1000, verbose=True):
    """
    Method for starting the play of a game to completion.
    -----------
    nmoves:
      The limit of how many moves (plies) to play unless the game ends on
      it's own first.
    verbose:
      Setting verbose=True displays additional text messages.
    """

    history = []

    if verbose:
        self.show()

    for self.nmove in range(1, nmoves + 1):

        if self.is_over():
            break

        move = self.player.ask_move(self)
        history.append((deepcopy(self), move))
        self.make_move(move)
        # print(f'board: {self.board}')1

        if verbose:
```

```
            print("\nMove #%d: player %d plays %s :" % (
                  self.nmove, self.nplayer, str(move)))
            self.show()

        self.switch_player()

    history.append(deepcopy(self))

    return history

@property
def nopponent(self):
    return 2 if (self.nplayer == 1) else 1

@property
def player(self):
    return self.players[self.nplayer - 1]

@property
def opponent(self):
    return self.players[self.nopponent - 1]

def possible_moves(self):
    return [i+1 for i, e in enumerate(self.board) if e == 0]

def make_move(self, move):
    self.board[int(move)-1] = self.nplayer
```

```
    def switch_player(self):

        self.nplayer = self.nopponent


    def copy(self):

        return deepcopy(self)


    def get_move(self):

        """

        Method for getting a move from the current player.

        """

        return self.player.ask_move(self)


    def play_move(self, move):

        """

        Method for playing one move with the current player.

        -----------

        move:

          The move to be played; should match an entry in the
'.possibles_moves()' list.

        """

        result = self.make_move(move)

        self.switch_player()

        return result


    def lose(self):

        """ Does the opponent have four in a row? """

        lose_lines = [[1, 2, 3, 4], [5, 6, 7, 8],
```

```
                     [9, 10, 11, 12], [13, 14, 15, 16],  # hor
                     [1, 5, 9, 13], [2, 6, 10, 14],
                     [3, 7, 11, 15], [4, 8, 12, 16],  # ver
                     [1, 6, 11, 16], [4, 7, 10, 13]]  # diag
        for line in lose_lines:
            for c in line:
                if all([(self.board[c-1] == self.nopponent) for c in line]):
                    return True
        return False

    def is_over(self):
        return (self.possible_moves() == []) or self.lose()

    def show(self):
        print('\n'+'\n'.join([
            ' '.join([['.', 'O', 'X'][self.board[4*j+i]]
                      for i in range(4)])
            for j in range(4)]))



if __name__ == "__main__":
    eval_func = EvalFunc2
    TTTPlus([AIPlayer(eval_func, max_depth=6), RealPlayer()]).play()
```

players.py

```
#!/usr/bin/python

#

# 4-in-a-row version of Tic Tac Toe two-player game

# author: Austin Perzben

# created: 2021-04-30


# Python3 program to demonstrate

# working of Alpha-Beta Pruning

# Initial values of Aplha and Beta

from evaluation_functions import EvalFunc1, EvalFunc2




class RealPlayer():

   def __init__(self, name='Human Player'):

       self.name = name


   def ask_move(self, game):

       possible_moves = game.possible_moves()

       # The str version of every move for comparison with the user input:

       possible_moves_str = list(map(str, game.possible_moves()))

       move = "NO_MOVE_DECIDED_YET"

       while True:

           move = input("\nPlayer %s what do you play ? " % (game.nplayer))

           if move == 'show moves':

               print("Possible moves:\n" + "\n".join(

                   ["#%d: %s" % (i+1, m) for i, m in
enumerate(possible_moves)])
```

```
                    + "\nType a move or type 'move #move_number' to play.")

            elif move == 'quit':
                raise KeyboardInterrupt

            elif move.startswith("move #"):
                # Fetch the corresponding move and return.
                move = possible_moves[int(move[6:])-1]
                return move

            elif str(move) in possible_moves_str:
                # Transform the move into its real type (integer, etc. and
return).
                move = possible_moves[possible_moves_str.index(str(move))]
                return move

    def __str__(self):
        return self.name



class AIPlayer():
    def __init__(self, eval_func, max_depth=9, name='AI Player'):
        self.name = name
        self.move = {}
        self.eval = eval_func
        self.depth = max_depth
```

```
    def negamax(self, game, depth, og_depth, alpha=float('-infinity'),
beta=float('infinity')):

        # Terminating condition. i.e
        # leaf node is reached
        if depth == 0 or game.is_over():
            score = self.eval(game)
            if score == 0:
                return score
            else:
                return (score - 0.01*depth*abs(score)/score)
            # print(f'score {score}')
            return score

        possible_moves = game.possible_moves()
        # print(f'moves {possible_moves}')
        state = game
        best_move = possible_moves[0]

        if depth == og_depth:
            state.ai_move = possible_moves[0]

        bestValue = float('-infinity')

        # Recur for left and right children
        for move in possible_moves:

            game = state.copy()  # re-initialize move
```

```
            game.make_move(move)
            game.switch_player()

            move_alpha = -self.negamax(game, depth-1, og_depth, -beta, -alpha)

            if bestValue < move_alpha:
                bestValue = move_alpha
                best_move = move

            if alpha < move_alpha:
                alpha = move_alpha
                if depth == og_depth:
                    state.ai_move = move
                if (alpha >= beta):
                    break
        return bestValue

    def ask_move(self, game):
        self.alpha = self.negamax(game, self.depth, self.depth)
        return game.ai_move

    def __str__(self):
        return self.name
```

evaluation_functions.py

```
#!/usr/bin/python
#
# 4-in-a-row version of Tic Tac Toe two-player game
# author: Austin Perzben
# created: 2021-04-30


def EvalFunc1(game):
    """
    Straight-forward evaluation function based simply
    on the game being lost at the current state or not.
    """
    return - 100 if game.lose() else 0


def EvalFunc2(game):
    """
    More specific evaluation function based on how many lines
    exist where the current player has an advantage. Advantage
    here means being the only player with marks on the line in question.

    A better evaluation function for Tic-Tac-Toe is:
        +100 for EACH 3-in-a-line for computer.
        +10 for EACH 2-in-a-line (with a empty cell) for computer.
        +1 for EACH 1-in-a-line (with two empty cells) for computer.
        Negative scores for opponent, i.e., -100, -10, -1 for EACH opponent's
3-in-a-line, 2-in-a-line and 1-in-a-line.
```

```
       0 otherwise (empty lines or lines with both computer's and opponent's
seed).

   Compute the scores for each of the 8 lines (3 rows, 3 columns and 2
diagonals) and obtain the sum.

   """

   score = 0

   rows = [game.board[i*4:i*4+4] for i in range(4)]

   cols = [game.board[i::4] for i in range(4)]

   diag = [game.board[0::5], game.board[3:13:3]]

   lines = rows + cols + diag

   for line in lines:

       x_taken = sum([1 for x in line if x == game.nplayer])

       x_lost = sum([1 for x in line if x == game.nopponent])

       x_free = sum([1 for x in line if x == 0])

       # print(line)

       # print(x_taken)

       # print(x_free)


       if x_lost == 1 and x_free == 3:

           score -= 1

       elif x_lost == 2 and x_free == 2:

           score -= 10

       elif x_lost == 3 and x_free == 1:

           score -= 100

       if x_taken == 1 and x_free == 3:

           score += 1

       elif x_taken == 2 and x_free == 2:

           score += 10
```

```
        elif x_taken == 3 and x_free == 1:

            score += 100

        else:

            score = 0

            break


    return score
```