adversarial search

November 10, 2022

1 Minimax Search

The Minimax algorithm is a relatively simple algorithm used for optimal decision-making in game theory and artificial intelligence. Again, since these algorithms heavily rely on being efficient, the vanilla algorithm's performance can be heavily improved by using alpha-beta pruning.

1.1 Minimax Implementation in Python

In the code below, we will be using an evaluation function that is fairly simple and common for all games in which it's possible to search the whole tree, all the way down to leaves.

It has 3 possible values:

- -1 if player that seeks minimum wins
- 0 if it's a tie
- 1 if player that seeks maximum wins

Since we'll be implementing this through a tic-tac-toe game, let's go through the building blocks.

[ ]:
```
# We'll use the time module to measure the time of evaluating
# game tree in every move. It's a nice way to show the
# distinction between the basic Minimax and Minimax with
# alpha-beta pruning :)
import time

class Game:
    def __init__(self):
        self.initialize_game()

    def initialize_game(self):
        self.current_state = [['.','.','.'],
                              ['.','.','.'],
                              ['.','.','.']]

        # Player X always plays first
        self.player_turn = 'X'

    # First, let's make a constructor and draw out the board:
    def draw_board(self):
        for i in range(0, 3):
            for j in range(0, 3):
                print('{}|'.format(self.current_state[i][j]), end=" ")
            print()
        print()


    # we need a way to check if a move is legal
    # Determines if the made move is a legal move
    def is_valid(self, px, py):
        if px < 0 or px > 2 or py < 0 or py > 2:
            return False
        elif self.current_state[px][py] != '.':
            return False
        else:
            return True

    # we need a simple way to check if the game has ended. In tic-tac-toe,
    # a player can win by connecting three consecutive symbols in either
    # a horizontal, diagonal or vertical line:

    # Checks if the game has ended and returns the winner in each case
    def is_end(self):
        # Vertical win
        for i in range(0, 3):
            if (self.current_state[0][i] != '.' and
                self.current_state[0][i] == self.current_state[1][i] and
                self.current_state[1][i] == self.current_state[2][i]):
                return self.current_state[0][i]

    # Horizontal win
        for i in range(0, 3):
            if (self.current_state[i] == ['X', 'X', 'X']):
                return 'X'
            elif (self.current_state[i] == ['O', 'O', 'O']):
                return 'O'

    # Main diagonal win
        if (self.current_state[0][0] != '.' and
            self.current_state[0][0] == self.current_state[1][1] and
            self.current_state[0][0] == self.current_state[2][2]):
            return self.current_state[0][0]

    # Second diagonal win
        if (self.current_state[0][2] != '.' and
            self.current_state[0][2] == self.current_state[1][1] and
            self.current_state[0][2] == self.current_state[2][0]):
            return self.current_state[0][2]
    # Is whole board full?
        for i in range(0, 3):
            for j in range(0, 3):
                # There's an empty field, we continue the game
                if (self.current_state[i][j] == '.'):
                    return None

    # It's a tie!
        return '.'

   # The AI we play against is seeking two things - to maximize its own score and to minimize ours.
   # To do that, we'll have a max() method that the AI uses for making optimal decisions.
   # Player 'O' is max, in this case AI

    def max(self):

       # Possible values for maxv are:
       # -1 - loss
       # 0  - a tie
       # 1  - win

       # We're initially setting it to -2 as worse than the worst case:
       maxv = -2

       px = None
       py = None

       result = self.is_end()

       # If the game came to an end, the function needs to return
       # the evaluation function of the end. That can be:
       # -1 - loss
       # 0  - a tie
       # 1  - win

       if result == 'X':
          return(-1, 0, 0)
       elif result == 'O':
          return(1, 0, 0)
       elif result == '.':
          return(0, 0, 0)

       for i in range(0, 3):
           for j in range(0, 3):
            if self.current_state[i][j] == '.':
                # On the empty field player 'O' makes a move and calls Min
                # That's one branch of the game tree.
                self.current_state[i][j] = 'O'
                (m, min_i, min_j) = self.min()
                # Fixing the maxv value if needed
                if m > maxv:
                    maxv = m
                    px = i
                    py = j
                # Setting back the field to empty
                self.current_state[i][j] = '.'
    return (maxv, px, py)

# We will also include a min() method that will serve as a helper for us to␣
↪minimize the AI's score:
# Player 'X' is min, in this case human

def min(self):
    # Possible values for minv are:
    # -1 - win
    # 0  - a tie
    # 1  - loss

    # We're initially setting it to 2 as worse than the worst case:
    minv = 2

    qx = None
    qy = None

    result = self.is_end()

    if result == 'X':
        return (-1, 0, 0)
    elif result == 'O':
        return (1, 0, 0)
    elif result == '.':
        return (0, 0, 0)

    for i in range(0, 3):
        for j in range(0, 3):
            if self.current_state[i][j] == '.':
                self.current_state[i][j] = 'X'
                (m, max_i, max_j) = self.max()
                if m < minv:
                    minv = m
                    qx = i
                    qy = j
                self.current_state[i][j] = '.'

        return (minv, qx, qy)
    # And ultimately, let's make a game loop that allows us to play against the␣
↪AI:

    def play(self):
        while True:
            self.draw_board()
            self.result = self.is_end()

            # Printing the appropriate message if the game has ended
            if self.result != None:
                if self.result == 'X':
                    print('The winner is X!')
                elif self.result == 'O':
                    print('The winner is O!')
                elif self.result == '.':
                    print("It's a tie!")

                self.initialize_game()
                return

            # If it's player's turn
            if self.player_turn == 'X':

                while True:

                    start = time.time()
                    (m, qx, qy) = self.min()
                    end = time.time()
                    print('Evaluation time: {}s'.format(round(end - start, 7)))
                    print('Recommended move: X = {}, Y = {}'.format(qx, qy))

                    px = int(input('Insert the X coordinate: '))
                    py = int(input('Insert the Y coordinate: '))

                    (qx, qy) = (px, py)

                    if self.is_valid(px, py):
                        self.current_state[px][py] = 'X'
                        self.player_turn = 'O'
                        break
                    else:
                        print('The move is not valid! Try again.')
            # If it's AI's turn
            else:
                (m, px, py) = self.max()
                self.current_state[px][py] = 'O'
                self.player_turn = 'X'
```

Let's start the game and take a look at what happens when we follow the recommended sequence of turns - i.e. we play optimally:

[ ]:
```
g = Game()
g.play()
```

As you've noticed, winning against this kind of AI is impossible. If we assume that both player and AI are playing optimally, the game will always be a tie. Since the AI always plays optimally, if we slip up, we'll lose.

Take a close look at the evaluation time, as we will compare it to the next, improved version of the algorithm in the next example.

1.2 Alpha-Beta Pruning

Alpha–beta is an improved minimax using a heuristic. It stops evaluating a move when it makes sure that it's worse than previously examined move. Such moves need not to be evaluated further.

When added to a simple minimax algorithm, it gives the same output, but cuts off certain branches that can't possibly affect the final decision - dramatically improving the performance.

The main concept is to maintain two values through whole search:

- Alpha: Best already explored option for player Max
- Beta: Best already explored option for player Min

Initially, alpha is negative infinity and beta is positive infinity, i.e. in our code we'll be using the worst possible scores for both players.

This method allows us to ignore many branches that lead to values that won't be of any help for our decision, nor they would affect it in any way.

With that in mind, let's modify the min() and max() methods from before in a derived class inherited from the previous Game class

[ ]:
```
class GamePruning(Game):

    #modified max method
    def max_alpha_beta(self, alpha, beta):
        maxv = -2
        px = None
        py = None

        result = self.is_end()

        if result == 'X':
            return (-1, 0, 0)
        elif result == 'O':
            return (1, 0, 0)
        elif result == '.':
            return (0, 0, 0)

        for i in range(0, 3):
            for j in range(0, 3):
                if self.current_state[i][j] == '.':
                    self.current_state[i][j] = 'O'
                    (m, min_i, in_j) = self.min_alpha_beta(alpha, beta)
                    if m > maxv:
                        maxv = m
                        px = i
                        py = j
                    self.current_state[i][j] = '.'

                    # Next two ifs in Max and Min are the only difference␣
↪between regular algorithm and minimax
                    if maxv >= beta:
                        return (maxv, px, py)

                    if maxv > alpha:
                        alpha = maxv

        return (maxv, px, py)


    # modified min method
    def min_alpha_beta(self, alpha, beta):

        minv = 2

        qx = None
        qy = None

        result = self.is_end()

        if result == 'X':
            return (-1, 0, 0)
        elif result == 'O':
            return (1, 0, 0)
        elif result == '.':
            return (0, 0, 0)

        for i in range(0, 3):
            for j in range(0, 3):
                if self.current_state[i][j] == '.':
                    self.current_state[i][j] = 'X'
                    (m, max_i, max_j) = self.max_alpha_beta(alpha, beta)
                    if m < minv:
                        minv = m
                        qx = i
                        qy = j
                    self.current_state[i][j] = '.'

                    if minv <= alpha:
                        return (minv, qx, qy)

                    if minv < beta:
                        beta = minv

        return (minv, qx, qy)

    # The game loop:
    def play_alpha_beta(self):

        while True:
            self.draw_board()
            self.result = self.is_end()

            if self.result != None:
                if self.result == 'X':
                    print('The winner is X!')
                elif self.result == 'O':
                    print('The winner is O!')
                elif self.result == '.':
                    print("It's a tie!")


                self.initialize_game()
                return

            if self.player_turn == 'X':

                while True:
                    start = time.time()
                    (m, qx, qy) = self.min_alpha_beta(-2, 2)
                    end = time.time()
                    print('Evaluation time: {}s'.format(round(end - start, 7)))
                    print('Recommended move: X = {}, Y = {}'.format(qx, qy))

                    px = int(input('Insert the X coordinate: '))
                    py = int(input('Insert the Y coordinate: '))
                qx = px
                qy = py

                if self.is_valid(px, py):
                    self.current_state[px][py] = 'X'
                    self.player_turn = 'O'
                    break
                else:
                    print('The move is not valid! Try again.')

        else:
            (m, px, py) = self.max_alpha_beta(-2, 2)
            self.current_state[px][py] = 'O'
            self.player_turn = 'X'
```

Playing the game is the same as before, though if we take a look at the time it takes for the AI to find optimal solutions, there's a big difference:

[ ]:
```
g2 = GamePruning()
g2.play()
```

After testing and starting the program from scratch for a few times, Write the results the table below:

Algorithm	Minimum time	Maximum time
Minimax		
Alpha-beta		

[ ]:
```
```