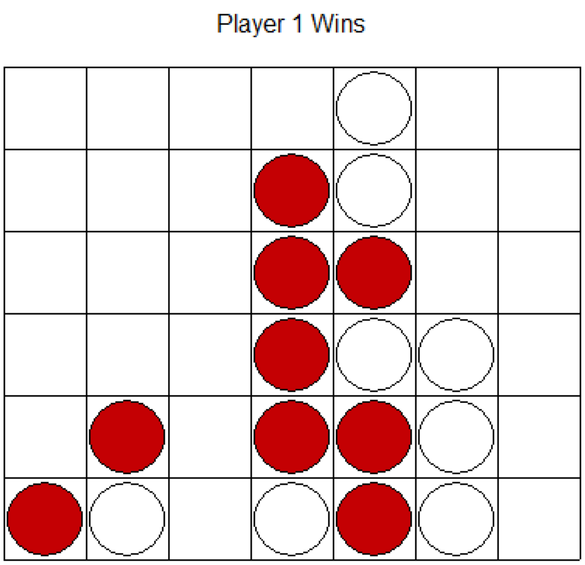
CPSC 4420/6420: Artificial Intelligence

Submitted By:
Charanjit Singh
Parampreet Singh

Homework Project 3 (due Wednseday, November 09)



In this programming assignment, you will design agents for the zero-sum game Connect Four, in which two players take turns dropping a colored disc from the top into a 7x6 grid. The game is won by forming a horizontal, vertical, or diagonal line of four same colored discs. See here for more details.

Basic Requirements

After downloading the attached source code, you should be able to display the game environment and select different game modes including Agent vs Agent, and Agent vs Human by typing the following at the command line:

```
python connect4.py
```

You can run the human-vs-human version to review how the game is played. Your task is to modify connect4.py in order to implement three of algorithms covered in class: minimax, alpha-beta pruning, and expectimax. Note that as different algorithms differ only in the dispatching mechanism, if you have minimax working correctly, implementing the rest of the methods should be straightforward.

Question 1 (4 points) Minimax

Modify the `minimax` function in `connect4.py` to implement depth-limited minimax search. As it's not feasible to search the entire game tree, your code should limit the search to an arbitrary depth by using the GUI. Score the leaves of your minimax tree with the supplied `evaluate` function in order to treat them as terminal nodes.

Ans 1) Click here for minimax code

Question 2 (3 points) Alpha-Beta Pruning

Modify the `alphabeta` function in `connect4.py` to use alpha-beta pruning and allow more efficiently exploration of the minimax tree. You should be able to see a speed-up as the depth of the tree increases.

Ans 2) Click here for minimax with alpha beta pruning code

Question 3 (3 points) Expectimax

Minimax and alpha-beta assume that MAX plays against an adversary who makes optimal decisions. Modify the `expectimax` function in `connect4.py` to model probabilistic behavior of opponents that may make suboptimal decisions. To do so, you will replace MIN nodes (Agent2 or Human) with chance nodes. To simplify your code, assume you will only be running against an adversary which chooses actions uniformly at random.

Ans 3) Click here for expectimax code

Important notes:

1. Depending on your GUI selection, the game can be played between two agents, or an agent and a human player, or an agent and a random player. When an agent needs to make its next move, it runs an adversarial search as a MAX player and its opponent is considered to be a MIN (or a CHANCE) player depending on the search algorithm that you run. A random player simply makes a random valid move.
2. The minimax algorithm always considers that the adversary tries to minimize the score of the MAX player that initiated the game search. The adversary never considers its own score at all during this process. Therefore, when evaluating the utilities of the nodes at the maximum tree depth, the evaluation should always be made from MAX's point of view.
3. The pesudocode provided in the slides only returns the best utility value. However, here, you need to select the policy, i.e., the action that is associated with this value. To do so, you should consider all valid actions for the MAX player at the root of the tree, and return the one that leads to the best value.
4. We have provided you with a decent evaluation function that allows for effective depth limited search. Feel free to edit the `evaluate` function, if you are able to come up with a better one. A better function means lower computation time and/or higher winning percentage as compared to ours for a fixed depth. In case you're up to the task, please include a brief ReadMe explaining your strategy.

Submission

Please submit your modified `connect4.py` file to Project/Assignment 3 on Canvas. You can work in pairs if you want to. If you do so, include both names in your report.

Getting Help

If you get stuck, please do not hesitate to contact the TA and the instructor for help, and stop by during office hours. We also encourage you to post questions and initiate discussions on Canvas and discord. Your colleagues are also there to help you.

```
1 # connect4.py
2 # ---------
3 # Licensing Information:  You are free to use or extend these projects for
4 # educational purposes provided that (1) you do not distribute or publish
5 # solutions, (2) you retain this notice, and (3) you provide clear
6 # attribution to Clemson University and the authors.
7 #
8 # Authors: Pei Xu (peix@g.clemson.edu) and Ioannis Karamouzas (ioannis@g.clemson.edu)
9 #
10
11 """
12 In this assignment, the task is to implement the minimax algorithm with depth
13 limit for a Connect-4 game.
14
15 To complete the assignment, you must finish these functions:
16     minimax (line 196), alphabeta (line 237), and expectimax (line 280)
17 in this file.
18
19 In the Connect-4 game, two players place discs in a 6-by-7 board one by one.
20 The discs will fall straight down and occupy the lowest available space of
21 the chosen column. The player wins if four of his or her discs are connected
22 in a line horizontally, vertically or diagonally.
23 See https://en.wikipedia.org/wiki/Connect_Four for more details about the game.
24
25 A Board() class is provided to simulate the game board.
26 It has the following properties:
27     b.rows          # number of rows of the game board
28     b.cols          # number of columns of the game
```

```
board
    b.PLAYER1       # an integer flag to represent the player 1
    b.PLAYER2       # an integer flag to represent the player 2
    b.EMPTY_SLOT    # an integer flag to represent an empty slot in the board;

and the following methods:
    b.terminal()            # check if the game is terminal
                            # terminal means draw or someone wins

    b.has_draw()            # check if the game is a draw

    w = b.who_wins()        # return the winner of the game or None if there
                            # is no winner yet
                            # w should be in [b.PLAYER1,b.PLAYER2, None]

    b.occupied(row, col)    # check if the slot at the specific location is
                            # occupied

    x = b.get(row, col)     # get the player occupying the given slot
                            # x should be in [b.PLAYER1, b.PLAYER2, b.EMPTY_SLOT]

    row = b.row(r)          # get the specific row of the game described using
                            # b.PLAYER1, b.PLAYER2 and b.EMPTY_SLOT

    col = b.column(r)       # get a specific column of the game board

    b.placeable(col)        # check if a disc can be
```

```
placed at the specific
                            # column

    b.place(player, col)    # place a disc at the specific column for player
        # raise ValueError if the specific column does not have available space

    new_board = b.clone()   # return a new board instance having the same
                            # disc placement with b

    str = b.dump()          # a string to describe the game board using
                            # b.PLAYER1, b.PLAYER2 and b.EMPTY_SLOT
Hints:
    1. Depth-limited Search
        We use depth-limited search in the current code. That is we
    stop the search forcefully, and perform evaluation directly not only when a
    terminal state is reached but also when the search reaches the specified
    depth.
    2. Game State
        Three elements decide the game state. The current board state, the
    player that needs to take an action (place a disc), and the current search
    depth (remaining depth).
    3. Evaluation Target
        The minimax algorithm always considers that the adversary tries to
    minimize the score of the max player, for whom the algorithm is called
    initially. The adversary never considers its own score at all during this
    process. Therefore, when evaluating nodes, the target should always be
    the max player.
```

```
    4. Search Result
        The pesudo code provided in the slides only returns the best utility value.
    However, in practice, we need to select the action that is associated with this
    value. Here, such action is specified as the column in which a disc should be
    placed for the max player. Therefore, for each search algorithm, you should
    consider all valid actions for the max player, and return the one that leads
    to the best value.

"""

# use math library if needed
import math

def get_child_boards(player, board):
    """
    Generate a list of successor boards obtained by placing a disc
    at the given board for a given player

    Parameters
    ----------
    player: board.PLAYER1 or board.PLAYER2
        the player that will place a disc on the board
    board: the current board instance

    Returns
    -------
    a list of (col, new_board) tuples,
    where col is the column in which a new disc is placed (left column has a 0 index),
    and new_board is the resulting board instance
    """
    res = []
    for c in range(board.cols):
        if board.placeable(c):
```

```
            tmp_board = board.clone()
            tmp_board.place(player, c)
            res.append((c, tmp_board))
    return res


def evaluate(player, board):
    """
    This is a function to evaluate the advantage of the specific player at the
    given game board.

    Parameters
    ----------
    player: board.PLAYER1 or board.PLAYER2
        the specific player
    board: the board instance

    Returns
    -------
    score: float
        a scalar to evaluate the advantage of the specific player at the given
        game board
    """
    adversary = board.PLAYER2 if player == board.PLAYER1 else board.PLAYER1
    # Initialize the value of scores
    # [s0, s1, s2, s3, --s4--]
    # s0 for the case where all slots are empty in a 4-slot segment
    # s1 for the case where the player occupies one slot in a 4-slot line, the rest are empty
    # s2 for two slots occupied
    # s3 for three
    # s4 for four
    score = [0]*5
    adv_score = [0]*5

    # Initialize the weights
    # [w0, w1, w2, w3, --w4--]
```

```
    # w0 for s0, w1 for s1, w2 for s2, w3 for s3
    # w4 for s4
    weights = [0, 1, 4, 16, 1000]

    # Obtain all 4-slot segments on the board
    seg = []
    invalid_slot = -1
    left_revolved = [
        [invalid_slot]*r + board.row(r) + \
        [invalid_slot]*(board.rows-1-r) for r in range(board.rows)
    ]
    right_revolved = [
        [invalid_slot]*(board.rows-1-r) + board.row(r) + \
        [invalid_slot]*r for r in range(board.rows)
    ]
    for r in range(board.rows):
        # row
        row = board.row(r)
        for c in range(board.cols-3):
            seg.append(row[c:c+4])
    for c in range(board.cols):
        # col
        col = board.col(c)
        for r in range(board.rows-3):
            seg.append(col[r:r+4])
    for c in zip(*left_revolved):
        # slash
        for r in range(board.rows-3):
            seg.append(c[r:r+4])
    for c in zip(*right_revolved):
        # backslash
        for r in range(board.rows-3):
            seg.append(c[r:r+4])
    # compute score
    for s in seg:
        if invalid_slot in s:
            continue
        if adversary not in s:
            score[s.count(player)] += 1
```

```python
        if player not in s:
            adv_score[s.count(adversary)] += 1
    reward = sum([s*w for s, w in zip(score, weights)])
    penalty = sum([s*w for s, w in zip(adv_score, weights)])
    return reward - penalty

#Depth limited minimax search

def minimax(player, board, depth_limit):

    """
    Minimax algorithm with limited search depth.

    Parameters
    ----------
    player: board.PLAYER1 or board.PLAYER2
        the player that needs to take an action (place a disc in the game)
    board: the current game board instance
    depth_limit: int
        the tree depth that the search algorithm needs to go further before stopping
    max_player: boolean

    Returns
    -------
    placement: int or None
        the column in which a disc should be placed for the specific player
        (counted from the most left as 0)
        None to give up the game
    """
    max_player = player
    next_player = board.PLAYER2 if player == board.PLAYER1 else board.PLAYER1
    placement = None

### Please finish the code below ##############################################
```

```
##############################################################################
    def value(player, board, depth_limit):
        if board.terminal() or depth_limit == 0:
            return (evaluate(player, board), None)
        elif player == max_player:
            return max_value(player, board, depth_limit)
        elif player == next_player:
            return min_value(player, board, depth_limit)

    def max_value(player, board, depth_limit):
        l = []
        for c, b in get_child_boards(player, board):
            v, a = value(next_player, b, depth_limit-1)
            l.append((v, c))
        return max(l)

    def min_value(player, board, depth_limit):
        l = []
        for c, b in get_child_boards(player,board):
            v, a = value(max_player, b, depth_limit-1)
            l.append((v, c))
        return min(l)

    score, placement = value(max_player, board, depth_limit)

##############################################################################
    return placement

#Alpha beta pruning of minimax search for efficiency
def alphabeta(player, board, depth_limit):
    """
    Minimax algorithm with alpha-beta pruning.

     Parameters
```

```
    ----------
    player: board.PLAYER1 or board.PLAYER2
        the player that needs to take an action (place a disc in the game)
    board: the current game board instance
    depth_limit: int
        the tree depth that the search algorithm needs to go further before stopping
    alpha: float
    beta: float
    max_player: boolean


    Returns
    -------
    placement: int or None
        the column in which a disc should be placed for the specific player
        (counted from the most left as 0)
        None to give up the game
    """
    max_player = player
    next_player = board.PLAYER2 if player == board.PLAYER1 else board.PLAYER1
    alpha = -math.inf
    beta = math.inf
    placement = None

### Please finish the code below ##############################################
###############################################################################
    def value(player, board, depth_limit, alpha, beta):
        if board.terminal():
            return (evaluate(player, board), None)
        elif depth_limit == 0:
            return (evaluate(player, board), None)
        elif player == max_player:
            return max_value(player, board, depth_limit, alpha, beta)
```

```
        elif player == next_player:
            return min_value(player, board, depth_limit, alpha, beta)

    def max_value(player, board, depth_limit, alpha, beta):
        l = []
        for c, b in get_child_boards(player, board):
            v, a = value(next_player, b, depth_limit-1, alpha, beta)
            if v >= beta:
                return (v, c)
            l.append((v, c))
            alpha = max(alpha, v)
        return max(l)

    def min_value(player, board, depth_limit, alpha, beta):
        l = []
        for c, b in get_child_boards(player,board):
            v, a = value(max_player, b, depth_limit-1, alpha, beta)
            if v <= alpha:
                return (v, c)
            l.append((v, c))
            beta = min(beta, v)
        return min(l)

    score, placement = value(max_player, board, depth_limit, -math.inf, math.inf)

##########################################################################################
    return placement

#Expectimax search with uniform chance of selecting available actions
def expectimax(player, board, depth_limit):
    """
    Expectimax algorithm.
    We assume that the adversary of the initial
```

```
player chooses actions
    uniformly at random.
    Say that it is the turn for Player 1 when the function is called initially,
    then, during search, Player 2 is assumed to pick actions uniformly at
    random.

    Parameters
    ----------
    player: board.PLAYER1 or board.PLAYER2
        the player that needs to take an action (place a disc in the game)
    board: the current game board instance
    depth_limit: int
        the tree depth that the search algorithm needs to go before stopping
    max_player: boolean

    Returns
    -------
    placement: int or None
        the column in which a disc should be placed for the specific player
        (counted from the most left as 0)
        None to give up the game
    """
    max_player = player
    next_player = board.PLAYER2 if player == board.PLAYER1 else board.PLAYER1
    placement = None

    ### Please finish the code below ##############################################
    ##############################################################################
    def value(player, board, depth_limit):
        if board.terminal() or depth_limit == 0:
            return (evaluate(player, board), None)
        elif player == max_player:
```

```
                return max_value(player, board, depth_limit)
            elif player == next_player:
                return exp_value(player, board, depth_limit)

    def prob_action(player, board):
        possible_Actions = 0
        for c in range(board.cols):
            if board.placeable(c):
                possible_Actions += 1
        return (1/possible_Actions)

    def max_value(player, board, depth_limit):
        l = []
        for c, b in get_child_boards(player, board):
            v, a = value(next_player, b, depth_limit - 1)
            l.append((v, c))
        return max(l)

    def exp_value(player, board, depth_limit):
        l = 0
        for c, b in get_child_boards(player, board):
            v, a = value(max_player, b, depth_limit - 1)
            l = prob_action(player, board)*v + l
        return (l, None)

    score, placement = value(max_player, board, depth_limit)

################################################################################
    return placement

if __name__ == "__main__":
    from utils.app import App
    import tkinter

```

```python
    algs = {
        "Minimax": minimax,
        "Alpha-beta pruning": alphabeta,
        "Expectimax": expectimax
    }

    root = tkinter.Tk()
    App(algs, root)
    root.mainloop()

```