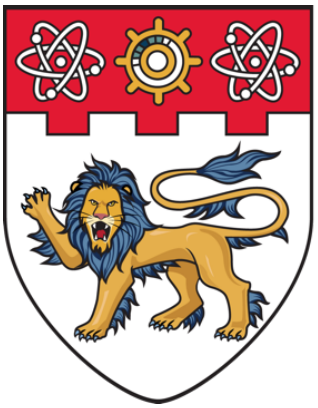

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

ASSIGNMENT2 REPORT

Hu Xiaoxiang
U1521319A
EEE

12 Oct, 2017

Contents

1 Overview

This algorithm includes a heuristic measure function and a minimax search with alpha-beta pruning. The heuristic evaluation function gives a weight on every move the computer makes. Higher the score is, more possible the decision is chose. Meanwhile, the alpha-beta pruning helps to reduce the search space so as to provide higher search efficiency.

2 Heuristic Evaluation Function

In tic-tac-toe, there are 3 conditions for a place, which 0 is defined for empty, 1 for offensive player's (X) stone and -1 for defensive player's (O) stone. For example, row '$_{\text{XO}}$' is represented by array [0, 1, -1], which means empty for first place, offensive player's stone at second place and defensive player's stone at third place. 9 different situations are defined by 4 rules in the heuristic evaluation function:

Given array [p1, p2, p3], where p1,p2,p3 represent 3 places of a raw.

Rule 1	p1+p2+p3 == 0

0 score is assigned if the row is empty or each of the player has 1 piece in a row. The sequence of the pieces is not important.

Rule 2	p1+p2+p3 == 3 or -3

3 pieces of one player are in a row, which represents the winning situation. The highest score 2000 or lowest score -2000 is assigned on this condition.

Rule 3	p1+p2+p3 == 2 or -2

2 pieces of one player are in a row. 100 or -100 score is assigned.

Rule 4	p1+p2+p3 == 1 or -1

There are 2 possibilities for this situation:

a. 1 piece of current player in a row

b. 2 pieces of opponent player and 1 piece of current player in a row

Obviously the first situation is more valuable to current player. Thus 10 score is assigned for the first situation and 0 is assigned for the second situation.

The total score is the sum of the 8 lines' score.

3 Minimax Searching Algorithm With Alpha-Beta Pruning

3.1 Search Strategy

In implementing minimax algorithm, each level in search space according to whose move it is at that point is labeld as Min or Max. For example, if the current player is offensive(X), it is labeled as Max and needs the maximum score of its children.

For alpha-beta pruning, each node has alpha and beta which represent the lower limit and upper limit respectively, e.p. [alpha, beta]. At Max node, alpha stores the current maximum value of its children and beta stores the current minimum value which is passed from its parent (Min). At Min node, beta stores the current minimum value of its children and alpha stores the current maxmium value which is passed from its parent (Max). Initially, -10000 (negative infinity) is assigned to the alpha of each Max node and 10000 (positive infinity) is assigned to the beta of each Min node.

From beginning, the search strategy keeps looking down until it reaches the node with maximum depth and calculates the score of that node based on the heuristic evaluation function. Next, it compares the score with the rest of nodes with the same parent and return the maximum or minimum value as alpha or beta according to whose move it is.

By repeating the previous two steps, the algorithm compares the value of alpha and beta continuously. The search can be stopped if the new beta of a Min node is less than or equal to its parent's alpha or the new alpha of a Max node is greater than or equal to its parent's beta.

3.2 Complexity Analysis

The average number of branch is defined as b and search depth is defined as d. The worst case is that all nodes need to be evaluated without any pruning. Thus the complexity is $\bigcirc(b^d)$.

For the best case, all the first player's moves must be evaluated to find the best one, but for each, only the best second player's move is needed, and the complexity is about $\bigcirc(b^{\frac{d}{2}})$

4 Advantages and Limitations

One of the benefits of alpha-beta pruning is that it reduces the search space significantly when comparing to a simple minimax algorithm. In terms of code complexity, minimax algorithm is also simpler than a rule-based algorithm which typically requires hundreds of rules to ensure a acceptable win rate.

The Limitation of minimax algorithm is that the search space still needs to be large enough to ensure the win rate when the specified rules are not sufficient. While if provided necessary rules, the search depth can be reduced correspondingly.

5 Win Situation

To be specific, if only rule 1 and 2 in the evaluation function are provided, the search depth needs to be at least 7 to play without losing. While if all of the 4 rules are given, then the search depth can be reduced to 2 or 3.

6 References

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

7 Appendix

```
def writeBoard(gameboard):
    gameGraph = []
    for i in range(3):
        _line = ""
        for j in range(3):
            if gameboard[i][j] == 0:
                _line += "_"
            elif gameboard[i][j] == 1:
                _line += "X"
            elif gameboard[i][j] == -1:
                _line += "O"
        gameGraph.append(_line)
    return gameGraph


# def readBoard(input_line):
#     line = list(input_line)
#     for n,i in enumerate(line):
#         if i == "_":
#             line[n] = 0
#         elif i == "X":
#             line[n] = 1
#         elif i == "O":
#             line[n] = -1
#     return line

# Heuristic Evaluation Func
def evalCurrentBoard(_cB):

    def evalLine(p1,p2,p3):
        scores = 0
        if p1+p2+p3 == 0:
            scores = 0
        elif p1+p2+p3 == 3:
            scores = 2000
        elif p1+p2+p3 == 2:
            scores = 100
```

```
            elif p1+p2+p3 == 1:
                if abs(p1)+abs(p2)+abs(p3) == 1:
                    scores = 10
                elif abs(p1)+abs(p2)+abs(p3) == 3:
                    scores = 0
            elif p1+p2+p3 == -3:
                scores = -2000
            elif p1+p2+p3 == -2:
                scores = -100
            elif p1+p2+p3 == -1:
                if abs(p1)+abs(p2)+abs(p3) == 1:
                    scores = -10
                elif abs(p1)+abs(p2)+abs(p3) == 3:
                    scores = 0
            return scores

        line_score = 0
        for r in range(3):
            line_score += evalLine(_cB[r][0],_cB[r][1],_cB[r][2])
        for c in range(3):
            line_score += evalLine(_cB[0][c],_cB[1][c],_cB[2][c])
        line_score += evalLine(_cB[0][0],_cB[1][1],_cB[2][2],)
        line_score += evalLine(_cB[2][0],_cB[1][1],_cB[0][2],)
        return line_score


    def strategyAnalysis(_currentBoard, _currentPlayer, _depth, previous_aorb, counter):

        def generateNewBoard(step):
            tempBoard = [row[:] for row in _currentBoard]
            if _currentPlayer == "X":
                tempBoard[step[0]][step[1]] = 1
            elif _currentPlayer == "O":
                tempBoard[step[0]][step[1]] = -1
            return tempBoard

        def checkWin():
            if abs(evalCurrentBoard(_currentBoard)) > 1000:
                return True
            else:
                return False

        nextPossibleMoves = []
        for r in range(3):
```

```
            for c in range(3):
                if _currentBoard[r][c] == 0:
                    nextPossibleMoves.append([r,c])

        if nextPossibleMoves == [] or _depth == -1 or checkWin():
            counter = counter + 1
            return [[], evalCurrentBoard(_currentBoard), previous_aorb, counter]
        else:
            # Minimax with alpha-beta pruning algorithm
            optimized_move = None
            if _currentPlayer == "X":
                _alpha = -10000  # Set local alpha for current Max and pass to next Min
                _beta = previous_aorb  # Assign previous Min to current beta
                for current_move in nextPossibleMoves:
                    # Only the second returned value is used as the current_move_score
                    return_array = (strategyAnalysis(generateNewBoard(current_move), "O",
                                                     _depth-1, _alpha, counter))
                    current_move_score = return_array[1]
                    counter = return_array[3]
                    if _alpha < current_move_score:
                        _alpha = current_move_score
                        optimized_move = current_move
                        if _alpha > _beta:
                            break
                return [optimized_move, _alpha, _beta, counter]
            elif _currentPlayer == "O":
                _beta = 10000  # Set local beta for current Min and pass to next Max
                _alpha = previous_aorb  # Assign previous Max to current alpha
                for current_move in nextPossibleMoves:
                    # Only the second returned value is used as the current_move_score
                    return_array = (strategyAnalysis(generateNewBoard(current_move), "X",
                                                     _depth-1, _beta, counter))
                    current_move_score = return_array[1]
                    counter = return_array[3]
                    if _beta > current_move_score:
                        _beta = current_move_score
                        optimized_move = current_move
                        if _alpha > _beta:
                            break
                return [optimized_move, _beta, _alpha, counter]


    def play(_cBoard, _cPlayer, _turns):
        # print("Turn {}".format(_turns))
```

```
    if _cPlayer == "X":
        nextStrategy = strategyAnalysis(_cBoard, _cPlayer, 8, 10000, 0)
        print(nextStrategy)
        nextStep = nextStrategy[0]
        if nextStep:
            _cBoard[nextStep[0]][nextStep[1]] = 1
            for i in writeBoard(_cBoard):
                print(i)
            # return play(_cBoard, "O", _turns+1)
            return _cBoard
        else:
            for i in writeBoard(_cBoard):
                print(i)
            return 1
    elif _cPlayer == "O":
        nextStrategy = strategyAnalysis(_cBoard, _cPlayer, 8, -10000, 0)
        print(nextStrategy)
        nextStep = nextStrategy[0]
        if nextStep:
            _cBoard[nextStep[0]][nextStep[1]] = -1
            for i in writeBoard(_cBoard):
                print(i)
            # return play(_cBoard, "X", _turns+1)
            return _cBoard
        else:
            return 1

turn = 1
newboard = [[0,0,0],
            [0,0,0],
            [0,0,0]]
player = input("Type in X or O, X is offensive, O is defensive:")
while turn < 9:
    if newboard == 1 or abs(evalCurrentBoard(newboard)) > 1000:
        print("Game Over!")
        break
    try:
        # print(turn)
        # check if player is offensive or defensive
        if player == 'X' or player == 'x':
            while True:
                next_decision = ([int(i) for i in
                input("Input next step m n(seperated by space m,n<=2):").split(' ')])
                if (newboard[next_decision[0]][next_decision[1]] == 0
```

```
                    and len(next_decision) == 2):
                    newboard[next_decision[0]][next_decision[1]] = 1
                    newboard = play(newboard, 'O', turn)
                    break
                else:
                    print("Current position is occupied!")
        elif (player == 'O' or player == 'o') and turn != 0:
            newboard = play(newboard, 'X', turn)
            if abs(evalCurrentBoard(newboard)) > 1000:
                print("Game Over!")
                break
            while True:
                next_decision = ([int(i) for i in
                input("Input next step m n(seperated by space m,n<=2):").split(' ')])
                if (newboard[next_decision[0]][next_decision[1]] == 0
                    and len(next_decision) == 2):
                    newboard[next_decision[0]][next_decision[1]] = -1
                    break
                else:
                    print("Current position is occupied!")
    except:
        pass

# currentBoard = [readBoard(input().strip()) for i in range(3)]
# play(currentBoard,player,1)
```