Depth of "Look-Ahead" Needed in Minimax Algorithm for an AI to Defeat a Human

Audrey McKnight

INTRODUCTION TO THE PROJECT / ABSTRACT

This independent project's goal was to create a Python program capable of defeating a human in the game "Ducks in A Row," a turn-based game in which the goal is to get your ducks arranged in a row of at least four by maneuvering them around the board. The author experimented with a specific part of the program, the "look-ahead", or depth of foresight that the Minimax Algorithm uses when calculating the optimal move, in order to discover the minimum amount of moves needed for an AI to consistently defeat the human, and see what the curve looks like for percentage of wins/losses. Due to computing power limitations, we were unable to test for a look-ahead beyond five moves.

AI IN GAMES

In the past 30 years, AI for playing structured games like Chess or Go have become popular. The code behind these games now utilizes advanced topics like Machine Learning, a process by which the AI can gain knowledge from its experiences and add to a database. The larger this database grows, the more effective the AI's moves will be. However, there are many other methods for making AI, however outdated they may be.

AI IN PYTHON

The ducks in a row AI is written in the coding language Python, utilizing matrices to represent a ducks in a row board, with the index [0,0] representing the upper left square of the board and [4,4] representing the lower right. Currently, gameplay goes as so: The board in its current state is drawn and you are given a prompt asking you which duck you want to move and to where. The AI then makes its move, calculated with the Minimax Algorithm, which

determines the optimal move to make in any situation. When the game detects four ducks in a row, it ends, declaring the winner.

HYPOTHESIS

This AI will be able to defeat humans consistently with depth of look-ahead (or foresight) 5 or 6. It may win against less-experienced players at lower depths, however, more experienced players will still be able to defeat it at least 90% of the time at these lower levels.

GAMEPLAY

The game of Ducks in a Row starts with a gameboard where each player has pieces in the arrangement depicted in Figure 1. Each player has six pieces. They can move in any direction but not jump over any other piece. Players alternate turns attempting to get 4 of their ducks in a row.

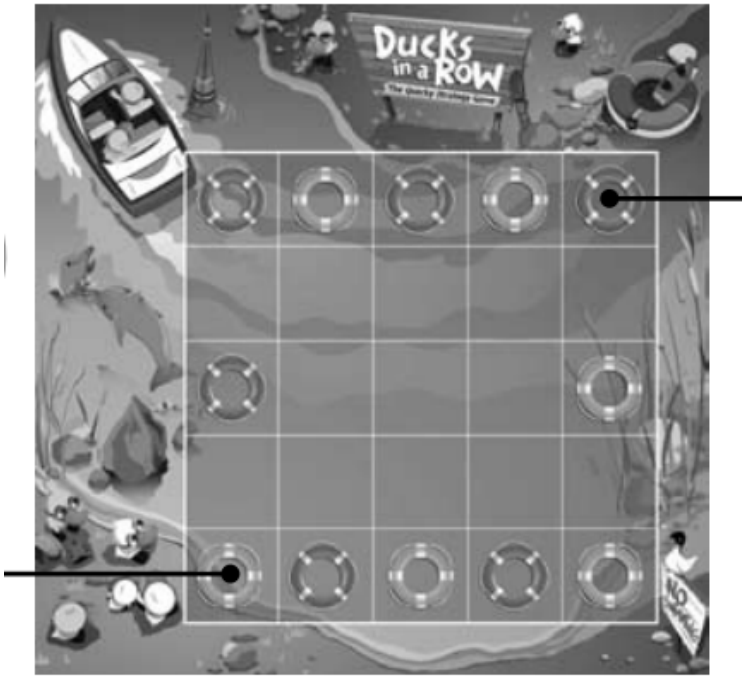


Figure 1, the starting position for Ducks in a Row (ThinkFun, 2008)

The program itself presents the AI in the form of a one-player game, human player versus the Row-Bot. The player can choose to move first or second, and as a difficulty buffer, the player can choose the number of moves that the Row-bot will look ahead, the depth of foresight. It is referred to in the actual program as the “look-ahead”.

Another important event the program checks for is whether or not the human player enters in a move that would move empty space or a duck that is not your color, a move that does not go in a straight or diagonal line of any length, or a move that requires for the duck to pass through another duck. If the program detects one of these rule infractions, your move is rejected and you are required to re-enter your desired move. The game will perform a check for whether not conditions for either player to win have been met, and if a player has, a winner will be declared and the program will end.

The basic gameplay will repeat this protocol for the most part, so far in this “completed” version of the program, no bugs in this basic protocol have been encountered. The current methods of checking for *some* of these game states are not as efficient as possible, too. I have recognized these small inefficiencies, and due to the deadline they will be neglected and finished in a later version. This is only the beta version. The Row-Bot’s final version will run the most efficient version of these checks in order to maximize speed due to the size of the calculations.

The number of possible moves for a five by five board using the Minimax Algorithm is large. For any given board position, there are about 30 legal moves. Then, each one of these legal moves results in about 30 possible responses. So, just to look at the Row-Bot’s move and the possible player responses, the program must evaluate about 900 board positions. While this

seems manageable, the number increases exponentially. If the program looks ahead 5 moves, we must evaluate 30^5 (over 24 million) positions.

THE CODE BEHIND THE GAME

The program uses the NumPy library for Python to represent the board as a matrix and maneuver and manipulate the ducks more easily, a decision made by suggestion of Matt McKnight, a Python expert whom the author consulted with for my initial plan and the best implementation of the minimax algorithm. One of the most important steps was getting the program to be able to understand the rules of the game, most importantly, being able to tell when a player's desired move is or is not legal. To do this the program uses a series of small algorithms, some of which are not as efficient as they could be, and could use some improvement. This will be addressed later in the paper.

Here are the shortened versions of the algorithms used to check whether or not the duck would be making a legal move:

The first step of the game is when the player gives the Row-bot the coordinates of the duck they want to move. These are saved as fromRow, and fromCol. The program checks whether or not there was a duck in that spot, then if there was a duck in the space they were moving to, the coordinates of which were saved as toRow and toCol. Next it checks between these two coordinate pairs to determine if the duck was moving, up or down, to the right or left, or diagonally. If the row's number did not change, it is moving to the right or left, which could be checked by determining whether or not the change between the columns was positive or negative. If the column's number did not change, the program can tell it was moving up/down, which it could check for in a similar manner to the way it checks between right/left.

If the human move was neither moving to right/left or up/down, then the program could check if the line between the two coordinates had a slope of one to determine whether or not it was a valid diagonal move. The program could determine if it was moving up/left, up/right, down/left, or down/right by checking whether or not the difference between the column numbers and the difference between the row numbers was greater or less than zero.

The program would also have to check whether or not the squares in the path of movement was empty. It does this by using a for loop that would add or subtract from the columns and/or rows and check if each new coordinate was an empty space or occupied by a duck.

THE MINIMAX ALGORITHM

The final step in coding this program was implementing the Minimax Algorithm (with alpha-beta pruning). The Minimax Algorithm is an algorithm used to determine the ideal move to make in turn-based games like Ducks in a Row. This algorithm was decided upon for use in this project for its relative simplicity when compared with other methods that can be used to make AI for board games.

The first step in coding the Minimax Algorithm is to develop a tree of all possible game states. This is done by looking at the current player and determining all of the legal moves that they player could make with the current board state. Then each move is evaluated to see if it is a winning move. Winning moves get a positive score. Moves which are indeterminate are given a score of zero. (One possible enhancement to the algorithm would be to run an evaluation function on boards that are not clear wins or losses, for example, favoring moves that would get

three ducks in a row over those that get two ducks in a row) The algorithm then takes each of these possible board states that would result from legal moves, and then looks at the moves the other player would make in response. When evaluating the other player's moves, winning moves get a negative score.

The algorithm then recursively traverses the tree, going down each branch. Since this program uses the Minimax Algorithm with alpha-beta pruning, this means that as soon as one part of the branch is identified as leading to a bad outcome (loss) we can close off all the paths that lead to it, and all the paths that stem from it. This decreases the time significantly. It filters through all these branches in this manner, until it has isolated the branches that lead to a win the shortest number of steps down the line. If after this process there are two or more branches that are "tied" for the shortest branch, it makes a random choice between them.

COMPETITION AGAINST HUMANS

The Ducks in a Row-Bot was unsuccessful in around 50% of its attempts at games against humans who were familiar with the mechanics of the game. This program could quite possibly increase this percentage considerably, however, with the strength of computing power that has been made available to me, the program caps the amount of moves that the Row-Bot looks ahead at three, as with any number greater than or equal to four the computing time becomes so long that any human player would quickly grow bored of the game, which would reasonably extend to at least 2 hours of play, with the AI taking around 15 or more minutes for each move.

CONCLUSION

Future optimization of the experiment could be achieved by streamlining the program's processes for absolute efficiency, and using the Google Cloud Platform for higher computing power. In addition to this, a neural network could be integrated in order to evaluate more precisely the quality of a move, rather than giving it 3 values, for the winning gamestate, the losing gamestate, and in-between. This in-between could be changed to prioritize some moves over others, while currently, it randomly chooses between them. All of these improvements would not only make the AI more effective and difficult for humans to compete against, but they would also speed up gameplay significantly, allowing for more trials in the experiment. (Though it is to be noted that if the more complicated gamestate evaluation were to be added without the aid of a neural network for added optimization processing time would be significantly longer.)

FULL CODE

```
import numpy
import random
import sys
import time


class Board:
    def __init__(self):
        # init is a function that runs whenever a new "board" is created
        self.matrix=numpy.zeros([5, 5], dtype=int)
        #Creating a 5by 5 matrix of 0s
    def startingposition(self):
        #this function puts all the ducks in their proper places
        #adding self as a parameter makes it so that anything that happens in
the method applies to a single board
        self.matrix[0, 0] = 1
        #1 represents green
        self.matrix[0, 1] = 2
        #2 represents orange
        self.matrix[0, 2] = 1
        self.matrix[0, 3] = 2
        self.matrix[0, 4] = 1
        self.matrix[1, 0] = 0
        #zero represents absence of ducks!
        self.matrix[1, 1] = 0
        self.matrix[1, 2] = 0
        self.matrix[1, 3] = 0
        self.matrix[1, 4] = 0
        self.matrix[2, 0] = 1
        self.matrix[2, 1] = 0
        self.matrix[2, 2] = 0
        self.matrix[2, 3] = 0
        self.matrix[2, 4] = 2
        self.matrix[3, 0] = 0
        self.matrix[3, 1] = 0
        self.matrix[3, 2] = 0
        self.matrix[3, 3] = 0
        self.matrix[3, 4] = 0
        self.matrix[4, 0] = 2
        self.matrix[4, 1] = 1
        self.matrix[4, 2] = 2
        self.matrix[4, 3] = 1
        self.matrix[4, 4] = 2

    def print(self):
        print(self.matrix)

    def isMoveValid(self, fromRow, fromCol, toRow, toCol, player):
        # make sure the right player is in the from space
        if self.matrix[fromRow, fromCol] != player:
```

```
            print("wrong player in from space: got {0} expected {1} in
({2},{3})".format(self.matrix[fromRow, fromCol], player, fromRow, fromCol))
            return False
        # make sure the space you are going to is empty
        if self.matrix[toRow,toCol] != 0:
            print("row you are moving to is not empty")
            return False
        #detect whether or not the duck is moving diagonally or straight
        if fromRow == toRow:
            # it's movin' straight to the side
            # make sure the spaces in between are empty
            for col in range(toCol,fromCol, numpy.sign(toCol-fromCol)):
                if self.matrix[fromRow, col] != 0:
                    print("something is in the way horizontal")
                    return False
        elif fromCol == toCol:
            # it's movin' up
            # make sure the spaces in between are empty
            for row in range(toRow,fromRow, numpy.sign(toRow-fromRow)):
                if self.matrix[row, fromCol] != 0:
                    print("something is in the way vertical")
                    return False
        else:
            # it's movin' diagonally
            # is it moving with slope 1
            if abs((toRow-fromRow)/(toCol-fromCol)) != 1:
                print("it's a bad slope")
                return False
            # make sure the spaces in between are empty
            # up and to the right
            if toCol-fromCol > 0 and toRow-fromRow > 0:
                col = fromCol
                for n in range(fromRow+1,toRow):
                    col += 1
                    #print(str(n) + ", " + str(col))
                    if self.matrix[n,col] != 0:
                        print("not empty up to the right")
                        return False
            #down and to the right
            if toCol-fromCol > 0 and toRow-fromRow < 0:
                col = fromCol
                for n in range(toRow, fromRow-1):
                    col += 1
                    if self.matrix[n, col] != 0:
                        print("not empty down to the right")
                        return False
            #moving up and left yo duh
            if fromCol-toCol > 0 and toRow-fromRow > 0:
                col = fromCol
                for n in range(fromRow+1, toRow):
                    col -= 1
                    if self.matrix[n, col] != 0:
```

```
                    print("not empty up and left")
                    return False
            #movign down and left
            if fromCol-toCol > 0 and toRow-fromRow < 0:
                col = fromCol
                for n in range(toRow, fromRow-1):
                    col -= 1
                    if self.matrix[n, col] != 0:
                        print("not empty down and left")
                        return False

        return True

    def moveDuck(self, fromRow, fromCol, toRow, toCol, player):
        self.matrix[fromRow, fromCol] = 0
        self.matrix[toRow, toCol] = player
        return True

    # return 1 if player 1 wins, 2 if player 2 wins, 0 if no player wins
    def check_win(self):
        # on the row
        for row in range(0, 5):
            # first four
            for player in range(1, 3):
                allsame = True
                for col in range(0, 4):
                    if self.matrix[row, col] != player:
                        allsame = False
                        break
                if allsame:
                    return player
            # last four
            for player in range(1, 3):
                allsame = True
                for col in range(1, 5):
                    if self.matrix[row, col] != player:
                        allsame = False
                        break
                if allsame:
                    return player
        # columns
        for col in range(0, 5):
            # top four
            for player in range(1, 3):
                allsame = True
                for row in range(0, 4):
                    if self.matrix[row, col] != player:
                        allsame = False
                        break
                if allsame:
                    return player
            # second four
```

```
            for player in range(1, 3):
                allsame = True
                for row in range(1, 5):
                    if self.matrix[row, col] != player:
                        allsame = False
                        break
                if allsame:
                    return player

        for player in range(1,3):
            # diagonal left -> right
            # main diag first four
            if [self.matrix[0, 0], self.matrix[1, 1], self.matrix[2, 2],
self.matrix[3, 3]] == [player, player, player, player]:
                return player
            # main diag second four
            if [self.matrix[1, 1], self.matrix[2, 2], self.matrix[3, 3],
self.matrix[4, 4]] == [player, player, player, player]:
                return player
            # lower diag
            if [self.matrix[1, 0], self.matrix[2, 1], self.matrix[3, 2],
self.matrix[4, 3]] == [player, player, player, player]:
                return player
            # upper diag
            if [self.matrix[0, 1], self.matrix[1, 2], self.matrix[2, 3],
self.matrix[3, 4]] == [player, player, player, player]:
                return player
            # diagonal right to left ->
            # main diagonal first four
            if [self.matrix[0, 4], self.matrix[1, 3], self.matrix[2, 2],
self.matrix[3, 1]] == [player, player, player, player]:
                return player
            # main diagonal second four
            if [self.matrix[1, 3], self.matrix[2, 2], self.matrix[3, 1],
self.matrix[4, 0]] == [player, player, player, player]:
                return player
            # lower diag
            if [self.matrix[0, 3], self.matrix[1, 2], self.matrix[2, 1],
self.matrix[3, 0]] == [player, player, player, player]:
                return player
            # upper diag
            if [self.matrix[1, 4], self.matrix[2, 3], self.matrix[3, 2],
self.matrix[4, 1]] == [player, player, player, player]:
                return player
        return 0

    # show all moves for a player
    def find_all_legal_moves(self, player):
        legal_moves = []
        for start_row in range(0, 5):
            for start_col in range(0, 5):
                if self.matrix[start_row, start_col] == player:
```

```
                    # find left moves
                    for move_col in range(start_col - 1, -1, -1):
                        if self.matrix[start_row][move_col] == 0:
                            legal_moves.append({"from_row": start_row,
"from_col": start_col,
                                                "to_row": start_row,
"to_col": move_col})
                            #print("({0},{1}) -> ({2},{3}) =
legal".format(start_row, start_col, start_row, move_col))
                        else:
                            break  # once you hit a piece you can go no
further in direction
                    # find right moves
                    for move_col in range(start_col + 1, 5):
                        if self.matrix[start_row][move_col] == 0:
                            legal_moves.append({"from_row": start_row,
"from_col": start_col,
                                                "to_row": start_row,
"to_col": move_col})
                            #print("({0},{1}) -> ({2},{3}) =
legal".format(start_row, start_col, start_row, move_col))
                        else:
                            break  # once you hit a piece you can go no
further in direction
                    # find up moves
                    for move_row in range(start_row - 1, -1, -1):
                        if self.matrix[move_row][start_col] == 0:
                            legal_moves.append({"from_row": start_row,
"from_col": start_col,
                                                "to_row": move_row,
"to_col": start_col})
                        else:
                            break
                    # find down moves
                    for move_row in range(start_row + 1, 5):
                        if self.matrix[move_row][start_col] == 0:
                            legal_moves.append({"from_row": start_row,
"from_col": start_col,
                                                "to_row": move_row,
"to_col": start_col})
                        else:
                            break
                    # find up-left moves
                    move_row = start_row - 1
                    move_col = start_col - 1
                    while (move_row >= 0) and (move_col >= 0):
                        if self.matrix[move_row][move_col] == 0:
                            legal_moves.append({"from_row": start_row,
"from_col": start_col,
                                                "to_row": move_row,
"to_col": move_col})
                        else:
```

```
                            break
                        move_row -= 1
                        move_col -= 1
                    # find up-right moves
                    move_row = start_row - 1
                    move_col = start_col + 1
                    while (move_row >= 0) and (move_col <= 4):
                        if self.matrix[move_row][move_col] == 0:
                            legal_moves.append({"from_row": start_row,
"from_col": start_col,
                                                "to_row": move_row, "to_col":
move_col})
                        else:
                            break
                        move_row -= 1
                        move_col += 1
                    # find down_left moves
                    move_row = start_row + 1
                    move_col = start_col - 1
                    while (move_row <= 4) and (move_col >= 0):
                        if self.matrix[move_row][move_col] == 0:
                            legal_moves.append({"from_row": start_row,
"from_col": start_col,
                                                "to_row": move_row, "to_col":
move_col})
                        else:
                            break
                        move_row += 1
                        move_col -= 1
                    # find down_right moves
                    move_row = start_row + 1
                    move_col = start_col + 1
                    while (move_row <= 4) and (move_col <= 4):
                        if self.matrix[move_row][move_col] == 0:
                            legal_moves.append({"from_row": start_row,
"from_col": start_col,
                                                "to_row": move_row, "to_col":
move_col})
                        else:
                            break
                        move_row += 1
                        move_col += 1
        return legal_moves



class Game:
    def __init__(self):
        self.myBoard = Board()
        self.myBoard.startingposition()
        self.human_player = 1
        self.computer_player = 2
```

```
        self.max_depth = 3

    def getHumanMove(self, player):
        possible = False
        while not possible:
            fromRow = int(input("Row of duck you want to move:"))
            fromColumn = int(input("Column of duck you want to move:"))
            if self.myBoard.matrix[fromRow, fromColumn] == player:
                possible = True
            else:
                print("You don't have a duck in that square")
        possible = False
        while not possible:
            toRow = int(input("Desired row you want to move that duck to:"))
            toColumn = int(input("Desired column you want to move that duck
to"))
            if self.myBoard.isMoveValid(fromRow, fromColumn, toRow, toColumn,
player):
                possible = True
            else:
                print(
                    "move from ({0},{1}) to ({2},{3}) is invalid for player
{4}".format(fromRow, fromColumn, toRow,

toColumn,

player))
        return fromRow, fromColumn, toRow, toColumn

    def nextPlayer(self, player):
        if player == 1:
            return 2
        return 1

    # def minimax(self, node, player):
    #     if node.complete():
    #         if node.X_won():
    #             return -1
    #         elif node.tied():
    #             return 0
    #         elif node.O_won():
    #             return 1
    #         best = None
    #     for move in node.find_all_legal_moves(player):
    #         node.make_move(move, player)
    #         val = self.minimax(node, self.nextPlayer(player), alpha, beta)
    #         node.make_move(move, None)
    #         if player == 'O':
    #             if val > best:
    #                 best = val
    #         else:
    #             if val < best:
```

```
    #                 best = val
    #         return best

    def alphabeta(self, node, player, alpha, beta, depth):
        winner = node.check_win()
        if winner == self.computer_player:
            return 1
        elif winner == self.human_player:
            return -1
        elif depth > self.max_depth:
            return 0
        for move in node.find_all_legal_moves(player):
            node.moveDuck(move["from_row"], move["from_col"], move["to_row"],
move["to_col"], player)
            val = self.alphabeta(node, self.nextPlayer(player), alpha, beta,
depth+1)
            # reverse the move
            node.moveDuck(move["to_row"], move["to_col"], move["from_row"],
move["from_col"], player)
            if player == self.computer_player:
                # find the best value!
                if val > alpha:
                    # the new best move
                    # alpha represents the ideal move, the one that will be
performed on the turn
                    alpha = val
                if alpha >= beta:
                    # beta is worst move
                    return beta
            else:
                # find where the human will do the worst
                if val < beta:
                    beta = val
                if beta <= alpha:
                    return alpha
        if player == self.computer_player:
            return alpha
        else:
            return beta

    def getRowbotMove(self, player):
        # legal_moves = self.myBoard.find_all_legal_moves(player)
        # move = random.choice(legal_moves)
        a = -2
        choices = []
        for move in self.myBoard.find_all_legal_moves(player):
            self.myBoard.moveDuck(move["from_row"], move["from_col"],
move["to_row"], move["to_col"], player)
            val = self.alphabeta(self.myBoard, self.nextPlayer(player), -2, 2,
0)
            # clears out move
```

```
            self.myBoard.moveDuck(move["to_row"], move["to_col"],
move["from_row"], move["from_col"], player)
            if val > a:
                a = val
                choices = [move]
            elif val == a:
                choices.append(move)
        move = random.choice(choices)
        return move["from_row"], move["from_col"], move["to_row"],
move["to_col"]

    def play(self):
        self.human_player = int(input("do you want to be player 1 or 2?"))
        self.computer_player = self.nextPlayer(self.human_player)
        self.myBoard.print()
        current_player = 1
        game_over = False
        while not game_over:
            if current_player == self.human_player:
                fromRow, fromColumn, toRow, toColumn =
self.getHumanMove(current_player)
            else:
                # uncomment for two human players
                # fromRow, fromColumn, toRow, toColumn =
self.getHumanMove(current_player)
                fromRow, fromColumn, toRow, toColumn =
self.getRowbotMove(current_player)
            if not self.myBoard.moveDuck(fromRow, fromColumn, toRow, toColumn,
current_player):
                print("Move from ({0},{1}) to ({2},{3}) was not
valid".format(fromRow, fromColumn, toRow, toColumn))
            else:
                print("moved from (" + str(fromRow) + "," + str(fromColumn) +
") to (" + str(toRow) + "," + str(toColumn) + ")")
                current_player = self.nextPlayer(current_player)
                self.myBoard.print()
            winner = self.myBoard.check_win()
            if winner > 0:
                print("Yeet, Player " + str(winner) + " wins and gets 2
pokedollars")
                game_over = True

sys.setrecursionlimit(2500)
print(sys.getrecursionlimit())
Game().play()
```

Sources

Lim, James. (2016) “Chess AI” https://github.com/lamesjim/Chess-AI

Norvig, Peter, & Russell, Stuart. (2005) *Artificial Intelligence: A Modern Approach.* http://aima.cs.berkeley.edu/

Payne, Trevor. (2014) “Let's Learn Python #21 - Min Max Algorithm”, https://www.youtube.com/watch?v=fInYh90YMJU

ThinkFun Games. (2008) “Ducks in Row Instructions” https://boardgamegeek.com/boardgame/34948/ducks-row

Woebker, Cecil. (2014) “Tic Tac Toe”. https://cwoebker.com/posts/tic-tac-toe