

|  |
| --- |
| **Title:** Implementation of Min-Max algorithms |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Objective:** Implementation of Min-Max algorithm

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **Course Outcome** | **After successful completion of the course students should be able to** |
| **CO2** | Analyse and solve problems for goal based agent architecture (searching and planning algorithms). |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **“Artificial Intelligence: a Modern Approach” by Russel and Norving, Pearson education Publications**
2. **“Artificial Intelligence” By Rich and knight, Tata Mcgraw Hill Publications**
3. [**www.cs.sfu.ca/CourseCentral/310/oschulte/mychapter5.pdf**](http://www.cs.sfu.ca/CourseCentral/310/oschulte/mychapter5.pdf)
4. [**http://cs.lmu.edu/~ray/notes/asearch/**](http://cs.lmu.edu/~ray/notes/asearch/)
5. **www.cs.cornell.edu/courses/cs4700/2011fa/.../06\_adversarialsearch.pdf**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Pre Lab/ Prior Concepts:** Two/Multi player Games and rules, state-space tree, searching algorithms and their analysis properties

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Historical Profile: -** The game playing has been integral part of human life. The multiplayer games are competitive environment in which everyone tries to gain more points for himself and wishes the opponent to gain minimum.

The game can be represented in form of a state space tree and one can follow the path from root to some goal node, for either of the player.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**New Concepts to be learned:** Adversarial search, minmax algorithm, minmax pruning,

**Adversarial Search:-**

In **computer science**, a **search algorithm** is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure.

***Adversarial search in Game playing:***

“In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.”

* But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.
* The environment with more than one agent is termed as **multi-agent environment**, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of another agent and the effect of that action on their performance.
* So, Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called **adversarial searches**, often known as Games.

**Assumptions and Aims**

* 2 agents whose actions alternate
* Utility values for each agent are the opposite of the other which creates adversarial situations.
* Fully observable environments.

**Applications in games**

* Must be a 2-player game.
* Players should alternate the moves.
* Both should have perfect information.
* Using dice is not involved.
* Clear rules for legal moves.

**How to strategize in each game**

* Consider all the legal moves you can make.
* Each move leads to a new board configuration.
* Evaluate each resulting position and determine which one is the best.
* Make that move.
* Wait for your opponent to move and repeat.

**Min-Max algorithm:**

* Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that the opponent is also playing optimally.
* Mini-Max algorithm uses recursion to search through the game-tree.
* The Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various two-player games. This Algorithm computes the minimax decision for the current state.
* In this algorithm two players play the game, one is called MAX and other is called MIN.
* Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
* Both Players of the game are opponents of each other, where MAX will select the maximized value and MIN will select the minimized value.
* The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
* The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

## Properties of Minimax algorithm

* **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
* **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
* **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **O(bm)**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
* **Space Complexity-** Space complexity of Minimax algorithm is also similar to DFS which is **O(bm).**

**Chosen Problem: Checkers GAME**

* Minimax is an artificial intelligence applied in two player games, such as tic-tac-toe, checkers, chess and go. These games are known as zero-sum games, because in a mathematical representation: one player wins (**+1**) and other player loses (**-1**) or both of anyone not to win (**0**).
* The algorithm search, recursively, the best move that leads the *Max* player to win or not lose (draw). It consider the current state of the game and the available moves at that state, then for each valid move it plays (alternating *min* and *max*) until it finds a terminal state (win, draw or lose).

**Code**

**checkers.py**

from copy *import* deepcopy

*import* time

*import* math

ansi\_black = "\u001b[30m"

ansi\_red = "\u001b[31m"

ansi\_green = "\u001b[32m"

ansi\_yellow = "\u001b[33m"

ansi\_blue = "\u001b[34m"

ansi\_magenta = "\u001b[35m"

ansi\_cyan = "\u001b[36m"

ansi\_white = "\u001b[37m"

ansi\_reset = "\u001b[0m"

class Node:

def \_\_init\_\_(*self*, *board*, *move*=None, *parent*=None, *value*=None):

self.board = board

self.value = value

self.move = move

self.parent = parent

def get\_children(*self*, *minimizing\_player*, *mandatory\_jumping*):

current\_state = deepcopy(self.board)

available\_moves = []

children\_states = []

big\_letter = ""

queen\_row = 0

*if* minimizing\_player is True:

available\_moves = Checkers.find\_available\_moves(current\_state, mandatory\_jumping)

big\_letter = "C"

queen\_row = 7

*else*:

available\_moves = Checkers.find\_player\_available\_moves(current\_state, mandatory\_jumping)

big\_letter = "B"

queen\_row = 0

*for* i in range(len(available\_moves)):

old\_i = available\_moves[i][0]

old\_j = available\_moves[i][1]

new\_i = available\_moves[i][2]

new\_j = available\_moves[i][3]

state = deepcopy(current\_state)

Checkers.make\_a\_move(state, old\_i, old\_j, new\_i, new\_j, big\_letter, queen\_row)

children\_states.append(Node(state, [old\_i, old\_j, new\_i, new\_j]))

*return* children\_states

def set\_value(*self*, *value*):

self.value = value

def get\_value(*self*):

*return* self.value

def get\_board(*self*):

*return* self.board

def get\_parent(*self*):

*return* self.parent

def set\_parent(*self*, *parent*):

self.parent = parent

class Checkers:

def \_\_init\_\_(*self*):

self.matrix = [[], [], [], [], [], [], [], []]

self.player\_turn = True

self.computer\_pieces = 12

self.player\_pieces = 12

self.available\_moves = []

self.mandatory\_jumping = False

*for* row in self.matrix:

*for* i in range(8):

row.append("---")

self.position\_computer()

self.position\_player()

def position\_computer(*self*):

*for* i in range(3):

*for* j in range(8):

*if* (i + j) % 2 == 1:

self.matrix[i][j] = ("c" + str(i) + str(j))

def position\_player(*self*):

*for* i in range(5, 8, 1):

*for* j in range(8):

*if* (i + j) % 2 == 1:

self.matrix[i][j] = ("b" + str(i) + str(j))

def print\_matrix(*self*):

i = 0

print()

*for* row in self.matrix:

print(i, end=" |")

i += 1

*for* elem in row:

print(elem, end=" ")

print()

print()

*for* j in range(8):

*if* j == 0:

j = " 0"

print(j, end=" ")

print("\n")

def get\_player\_input(*self*):

available\_moves = Checkers.find\_player\_available\_moves(self.matrix, self.mandatory\_jumping)

*if* len(available\_moves) == 0:

*if* self.computer\_pieces > self.player\_pieces:

print(

ansi\_red + "You have no moves left, and you have fewer pieces than the computer.YOU LOSE!" + ansi\_reset)

exit()

*else*:

print(ansi\_yellow + "You have no available moves.\nGAME ENDED!" + ansi\_reset)

exit()

self.player\_pieces = 0

self.computer\_pieces = 0

*while* True:

coord1 = input("Which piece[i,j]: ")

*if* coord1 == "":

print(ansi\_cyan + "Game ended!" + ansi\_reset)

exit()

*elif* coord1 == "s":

print(ansi\_cyan + "You surrendered.\nCoward." + ansi\_reset)

exit()

coord2 = input("Where to[i,j]:")

*if* coord2 == "":

print(ansi\_cyan + "Game ended!" + ansi\_reset)

exit()

*elif* coord2 == "s":

print(ansi\_cyan + "You surrendered.\nCoward." + ansi\_reset)

exit()

old = coord1.split(",")

new = coord2.split(",")

*if* len(old) != 2 or len(new) != 2:

print(ansi\_red + "Illegal input" + ansi\_reset)

*else*:

old\_i = old[0]

old\_j = old[1]

new\_i = new[0]

new\_j = new[1]

*if* not old\_i.isdigit() or not old\_j.isdigit() or not new\_i.isdigit() or not new\_j.isdigit():

print(ansi\_red + "Illegal input" + ansi\_reset)

*else*:

move = [int(old\_i), int(old\_j), int(new\_i), int(new\_j)]

*if* move not in available\_moves:

print(ansi\_red + "Illegal move!" + ansi\_reset)

*else*:

Checkers.make\_a\_move(self.matrix, int(old\_i), int(old\_j), int(new\_i), int(new\_j), "B", 0)

*for* m in range(8):

*for* n in range(8):

*if* self.matrix[m][n][0] == "c" or self.matrix[m][n][0] == "C":

self.computer\_pieces += 1

*elif* self.matrix[m][n][0] == "b" or self.matrix[m][n][0] == "B":

self.player\_pieces += 1

*break*

@staticmethod

def find\_available\_moves(*board*, *mandatory\_jumping*):

available\_moves = []

available\_jumps = []

*for* m in range(8):

*for* n in range(8):

*if* board[m][n][0] == "c":

*if* Checkers.check\_moves(board, m, n, m + 1, n + 1):

available\_moves.append([m, n, m + 1, n + 1])

*if* Checkers.check\_moves(board, m, n, m + 1, n - 1):

available\_moves.append([m, n, m + 1, n - 1])

*if* Checkers.check\_jumps(board, m, n, m + 1, n - 1, m + 2, n - 2):

available\_jumps.append([m, n, m + 2, n - 2])

*if* Checkers.check\_jumps(board, m, n, m + 1, n + 1, m + 2, n + 2):

available\_jumps.append([m, n, m + 2, n + 2])

*elif* board[m][n][0] == "C":

*if* Checkers.check\_moves(board, m, n, m + 1, n + 1):

available\_moves.append([m, n, m + 1, n + 1])

*if* Checkers.check\_moves(board, m, n, m + 1, n - 1):

available\_moves.append([m, n, m + 1, n - 1])

*if* Checkers.check\_moves(board, m, n, m - 1, n - 1):

available\_moves.append([m, n, m - 1, n - 1])

*if* Checkers.check\_moves(board, m, n, m - 1, n + 1):

available\_moves.append([m, n, m - 1, n + 1])

*if* Checkers.check\_jumps(board, m, n, m + 1, n - 1, m + 2, n - 2):

available\_jumps.append([m, n, m + 2, n - 2])

*if* Checkers.check\_jumps(board, m, n, m - 1, n - 1, m - 2, n - 2):

available\_jumps.append([m, n, m - 2, n - 2])

*if* Checkers.check\_jumps(board, m, n, m - 1, n + 1, m - 2, n + 2):

available\_jumps.append([m, n, m - 2, n + 2])

*if* Checkers.check\_jumps(board, m, n, m + 1, n + 1, m + 2, n + 2):

available\_jumps.append([m, n, m + 2, n + 2])

*if* mandatory\_jumping is False:

available\_jumps.extend(available\_moves)

*return* available\_jumps

*elif* mandatory\_jumping is True:

*if* len(available\_jumps) == 0:

*return* available\_moves

*else*:

*return* available\_jumps

@staticmethod

def check\_jumps(*board*, *old\_i*, *old\_j*, *via\_i*, *via\_j*, *new\_i*, *new\_j*):

*if* new\_i > 7 or new\_i < 0:

*return* False

*if* new\_j > 7 or new\_j < 0:

*return* False

*if* board[via\_i][via\_j] == "---":

*return* False

*if* board[via\_i][via\_j][0] == "C" or board[via\_i][via\_j][0] == "c":

*return* False

*if* board[new\_i][new\_j] != "---":

*return* False

*if* board[old\_i][old\_j] == "---":

*return* False

*if* board[old\_i][old\_j][0] == "b" or board[old\_i][old\_j][0] == "B":

*return* False

*return* True

@staticmethod

def check\_moves(*board*, *old\_i*, *old\_j*, *new\_i*, *new\_j*):

*if* new\_i > 7 or new\_i < 0:

*return* False

*if* new\_j > 7 or new\_j < 0:

*return* False

*if* board[old\_i][old\_j] == "---":

*return* False

*if* board[new\_i][new\_j] != "---":

*return* False

*if* board[old\_i][old\_j][0] == "b" or board[old\_i][old\_j][0] == "B":

*return* False

*if* board[new\_i][new\_j] == "---":

*return* True

@staticmethod

def calculate\_heuristics(*board*):

result = 0

mine = 0

opp = 0

*for* i in range(8):

*for* j in range(8):

*if* board[i][j][0] == "c" or board[i][j][0] == "C":

mine += 1

*if* board[i][j][0] == "c":

result += 5

*if* board[i][j][0] == "C":

result += 10

*if* i == 0 or j == 0 or i == 7 or j == 7:

result += 7

*if* i + 1 > 7 or j - 1 < 0 or i - 1 < 0 or j + 1 > 7:

*continue*

*if* (board[i + 1][j - 1][0] == "b" or board[i + 1][j - 1][0] == "B") and board[i - 1][

j + 1] == "---":

result -= 3

*if* (board[i + 1][j + 1][0] == "b" or board[i + 1][j + 1] == "B") and board[i - 1][j - 1] == "---":

result -= 3

*if* board[i - 1][j - 1][0] == "B" and board[i + 1][j + 1] == "---":

result -= 3

*if* board[i - 1][j + 1][0] == "B" and board[i + 1][j - 1] == "---":

result -= 3

*if* i + 2 > 7 or i - 2 < 0:

*continue*

*if* (board[i + 1][j - 1][0] == "B" or board[i + 1][j - 1][0] == "b") and board[i + 2][

j - 2] == "---":

result += 6

*if* i + 2 > 7 or j + 2 > 7:

*continue*

*if* (board[i + 1][j + 1][0] == "B" or board[i + 1][j + 1][0] == "b") and board[i + 2][

j + 2] == "---":

result += 6

*elif* board[i][j][0] == "b" or board[i][j][0] == "B":

opp += 1

*return* result + (mine - opp) \* 1000

@staticmethod

def find\_player\_available\_moves(*board*, *mandatory\_jumping*):

available\_moves = []

available\_jumps = []

*for* m in range(8):

*for* n in range(8):

*if* board[m][n][0] == "b":

*if* Checkers.check\_player\_moves(board, m, n, m - 1, n - 1):

available\_moves.append([m, n, m - 1, n - 1])

*if* Checkers.check\_player\_moves(board, m, n, m - 1, n + 1):

available\_moves.append([m, n, m - 1, n + 1])

*if* Checkers.check\_player\_jumps(board, m, n, m - 1, n - 1, m - 2, n - 2):

available\_jumps.append([m, n, m - 2, n - 2])

*if* Checkers.check\_player\_jumps(board, m, n, m - 1, n + 1, m - 2, n + 2):

available\_jumps.append([m, n, m - 2, n + 2])

*elif* board[m][n][0] == "B":

*if* Checkers.check\_player\_moves(board, m, n, m - 1, n - 1):

available\_moves.append([m, n, m - 1, n - 1])

*if* Checkers.check\_player\_moves(board, m, n, m - 1, n + 1):

available\_moves.append([m, n, m - 1, n + 1])

*if* Checkers.check\_player\_jumps(board, m, n, m - 1, n - 1, m - 2, n - 2):

available\_jumps.append([m, n, m - 2, n - 2])

*if* Checkers.check\_player\_jumps(board, m, n, m - 1, n + 1, m - 2, n + 2):

available\_jumps.append([m, n, m - 2, n + 2])

*if* Checkers.check\_player\_moves(board, m, n, m + 1, n - 1):

available\_moves.append([m, n, m + 1, n - 1])

*if* Checkers.check\_player\_jumps(board, m, n, m + 1, n - 1, m + 2, n - 2):

available\_jumps.append([m, n, m + 2, n - 2])

*if* Checkers.check\_player\_moves(board, m, n, m + 1, n + 1):

available\_moves.append([m, n, m + 1, n + 1])

*if* Checkers.check\_player\_jumps(board, m, n, m + 1, n + 1, m + 2, n + 2):

available\_jumps.append([m, n, m + 2, n + 2])

*if* mandatory\_jumping is False:

available\_jumps.extend(available\_moves)

*return* available\_jumps

*elif* mandatory\_jumping is True:

*if* len(available\_jumps) == 0:

*return* available\_moves

*else*:

*return* available\_jumps

@staticmethod

def check\_player\_moves(*board*, *old\_i*, *old\_j*, *new\_i*, *new\_j*):

*if* new\_i > 7 or new\_i < 0:

*return* False

*if* new\_j > 7 or new\_j < 0:

*return* False

*if* board[old\_i][old\_j] == "---":

*return* False

*if* board[new\_i][new\_j] != "---":

*return* False

*if* board[old\_i][old\_j][0] == "c" or board[old\_i][old\_j][0] == "C":

*return* False

*if* board[new\_i][new\_j] == "---":

*return* True

@staticmethod

def check\_player\_jumps(*board*, *old\_i*, *old\_j*, *via\_i*, *via\_j*, *new\_i*, *new\_j*):

*if* new\_i > 7 or new\_i < 0:

*return* False

*if* new\_j > 7 or new\_j < 0:

*return* False

*if* board[via\_i][via\_j] == "---":

*return* False

*if* board[via\_i][via\_j][0] == "B" or board[via\_i][via\_j][0] == "b":

*return* False

*if* board[new\_i][new\_j] != "---":

*return* False

*if* board[old\_i][old\_j] == "---":

*return* False

*if* board[old\_i][old\_j][0] == "c" or board[old\_i][old\_j][0] == "C":

*return* False

*return* True

def evaluate\_states(*self*):

t1 = time.time()

current\_state = Node(deepcopy(self.matrix))

first\_computer\_moves = current\_state.get\_children(True, self.mandatory\_jumping)

*if* len(first\_computer\_moves) == 0:

*if* self.player\_pieces > self.computer\_pieces:

print(

ansi\_yellow + "Computer has no available moves left, and you have more pieces left.\nYOU WIN!" + ansi\_reset)

exit()

*else*:

print(ansi\_yellow + "Computer has no available moves left.\nGAME ENDED!" + ansi\_reset)

exit()

dict = {}

*for* i in range(len(first\_computer\_moves)):

child = first\_computer\_moves[i]

value = Checkers.minimax(child.get\_board(), 4, -math.inf, math.inf, False, self.mandatory\_jumping)

dict[value] = child

*if* len(dict.keys()) == 0:

print(ansi\_green + "Computer has cornered itself.\nYOU WIN!" + ansi\_reset)

exit()

new\_board = dict[max(dict)].get\_board()

move = dict[max(dict)].move

self.matrix = new\_board

t2 = time.time()

diff = t2 - t1

print("Computer has moved (" + str(move[0]) + "," + str(move[1]) + ") to (" + str(move[2]) + "," + str(

move[3]) + ").")

print("It took him " + str(diff) + " seconds.")

@staticmethod

def minimax(*board*, *depth*, *alpha*, *beta*, *maximizing\_player*, *mandatory\_jumping*):

*if* depth == 0:

*return* Checkers.calculate\_heuristics(board)

current\_state = Node(deepcopy(board))

*if* maximizing\_player is True:

max\_eval = -math.inf

*for* child in current\_state.get\_children(True, mandatory\_jumping):

ev = Checkers.minimax(child.get\_board(), depth - 1, alpha, beta, False, mandatory\_jumping)

max\_eval = max(max\_eval, ev)

alpha = max(alpha, ev)

*if* beta <= alpha:

*break*

current\_state.set\_value(max\_eval)

*return* max\_eval

*else*:

min\_eval = math.inf

*for* child in current\_state.get\_children(False, mandatory\_jumping):

ev = Checkers.minimax(child.get\_board(), depth - 1, alpha, beta, True, mandatory\_jumping)

min\_eval = min(min\_eval, ev)

beta = min(beta, ev)

*if* beta <= alpha:

*break*

current\_state.set\_value(min\_eval)

*return* min\_eval

@staticmethod

def make\_a\_move(*board*, *old\_i*, *old\_j*, *new\_i*, *new\_j*, *big\_letter*, *queen\_row*):

letter = board[old\_i][old\_j][0]

i\_difference = old\_i - new\_i

j\_difference = old\_j - new\_j

*if* i\_difference == -2 and j\_difference == 2:

board[old\_i + 1][old\_j - 1] = "---"

*elif* i\_difference == 2 and j\_difference == 2:

board[old\_i - 1][old\_j - 1] = "---"

*elif* i\_difference == 2 and j\_difference == -2:

board[old\_i - 1][old\_j + 1] = "---"

*elif* i\_difference == -2 and j\_difference == -2:

board[old\_i + 1][old\_j + 1] = "---"

*if* new\_i == queen\_row:

letter = big\_letter

board[old\_i][old\_j] = "---"

board[new\_i][new\_j] = letter + str(new\_i) + str(new\_j)

def play(*self*):

print(ansi\_cyan + "##### WELCOME TO CHECKERS ####" + ansi\_reset)

print("\nSome basic rules:")

print("1.You enter the coordinates in the form i,j.")

print("2.You can quit the game at any time by pressing enter.")

print("3.You can surrender at any time by pressing 's'.")

print("Now that you've familiarized yourself with the rules, enjoy!")

*while* True:

answer = input("\nFirst, we need to know, is jumping mandatory?[Y/n]: ")

*if* answer == "Y" or answer == "y":

self.mandatory\_jumping = True

*break*

*elif* answer == "N" or answer == "n":

self.mandatory\_jumping = False

*break*

*elif* answer == "":

print(ansi\_cyan + "Game ended!" + ansi\_reset)

exit()

*elif* answer == "s":

print(ansi\_cyan + "You've surrendered before the game even started.\nPathetic." + ansi\_reset)

exit()

*else*:

print(ansi\_red + "Illegal input!" + ansi\_reset)

*while* True:

self.print\_matrix()

*if* self.player\_turn is True:

print(ansi\_cyan + "\nPlayer's turn." + ansi\_reset)

self.get\_player\_input()

*else*:

print(ansi\_cyan + "Computer's turn." + ansi\_reset)

print("Thinking...")

self.evaluate\_states()

*if* self.player\_pieces == 0:

self.print\_matrix()

print(ansi\_red + "You have no pieces left.\nYOU LOSE!" + ansi\_reset)

exit()

*elif* self.computer\_pieces == 0:

self.print\_matrix()

print(ansi\_green + "Computer has no pieces left.\nYOU WIN!" + ansi\_reset)

exit()

*elif* self.computer\_pieces - self.player\_pieces == 7:

wish = input("You have 7 pieces fewer than your opponent.Do you want to surrender?")

*if* wish == "" or wish == "yes":

print(ansi\_cyan + "Coward." + ansi\_reset)

exit()

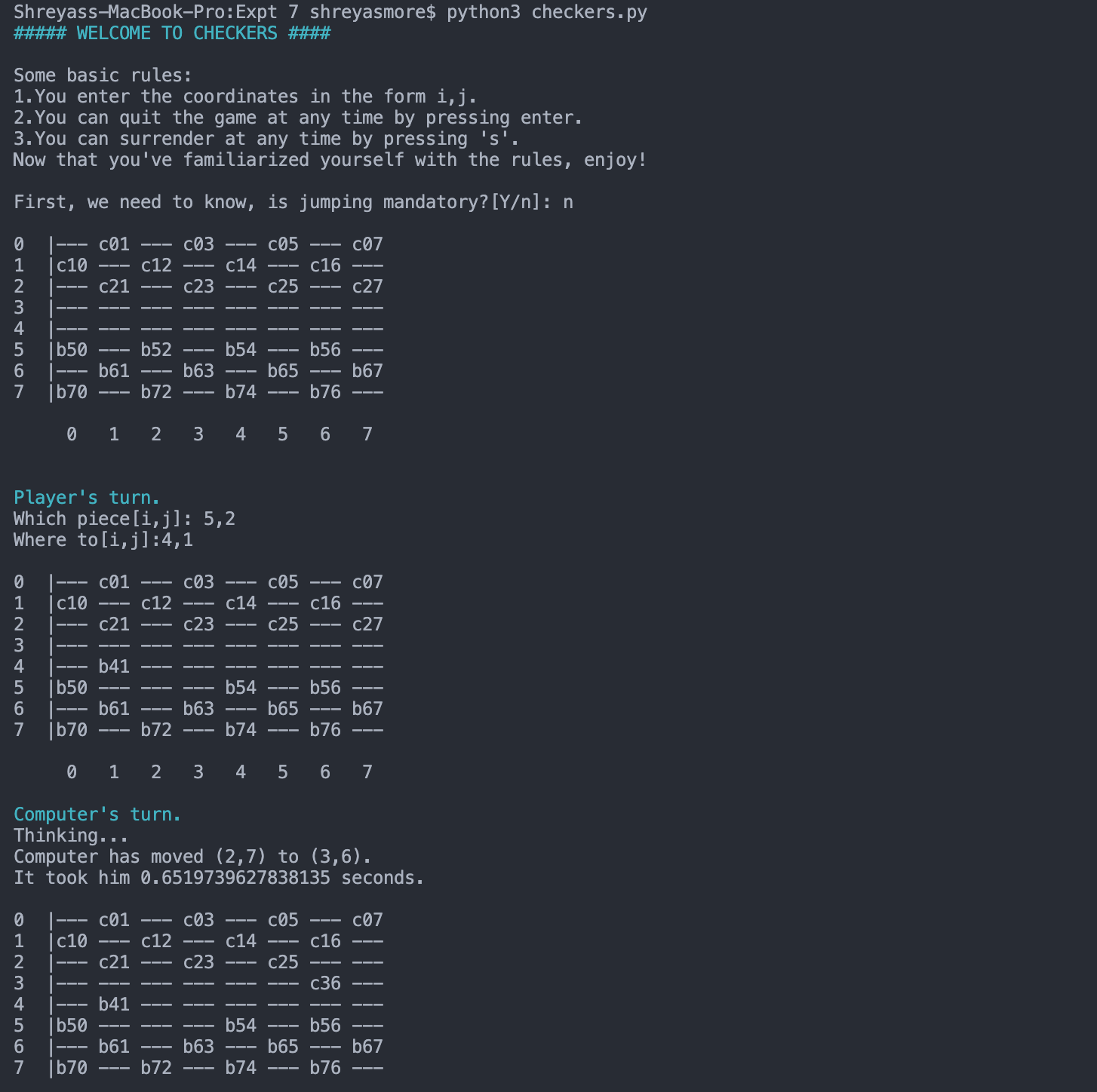
self.player\_turn = not self.player\_turn

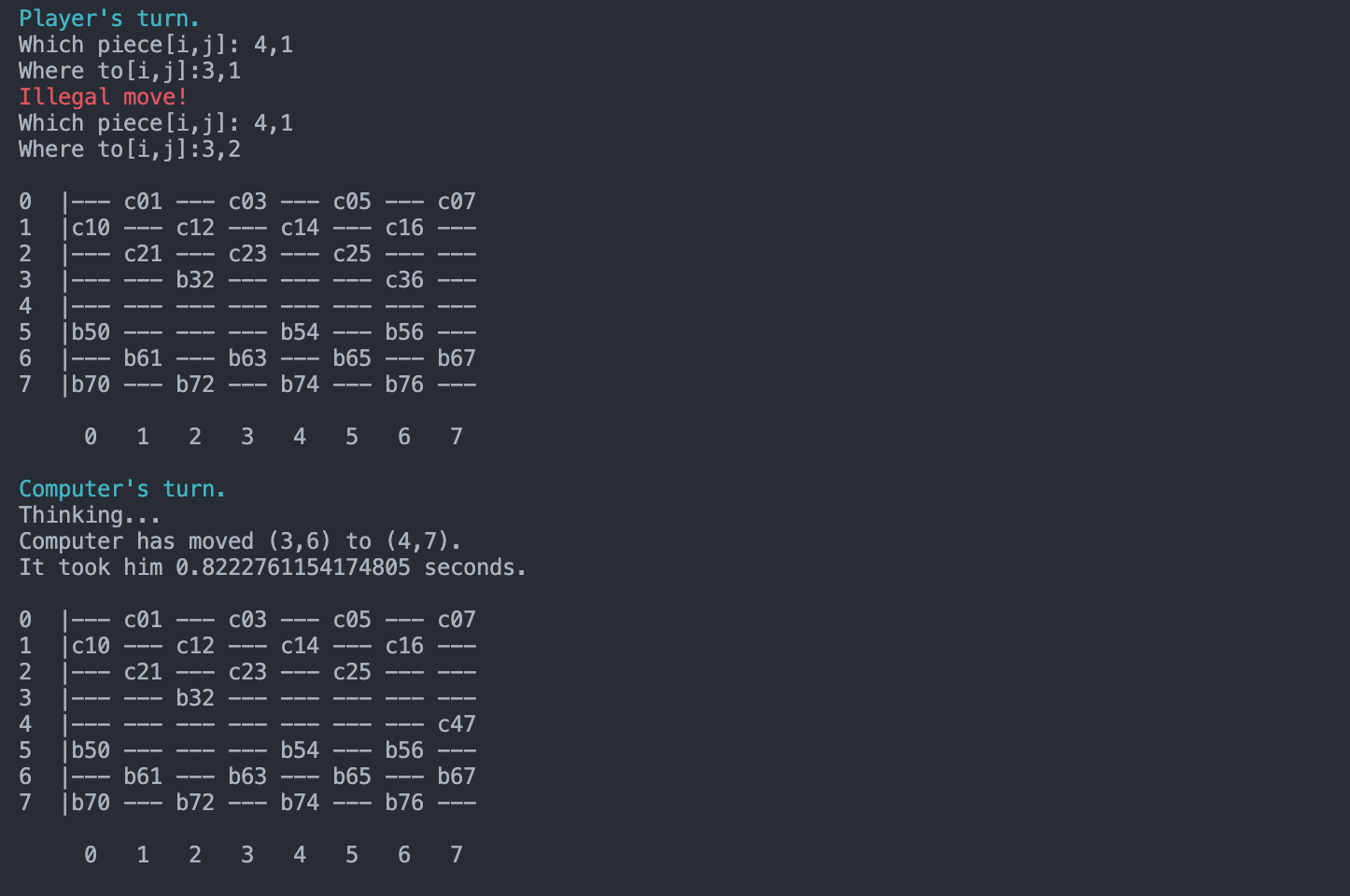
*if* \_\_name\_\_ == '\_\_main\_\_':

checkers = Checkers()

checkers.play()

**OUTPUT SCREENSHOTS**

****

****

**Post Lab objective Questions:**

1. **Which search is equal to minmax search but eliminates the branches that can’t influence the final decision?**
   1. Breadth-first search
   2. Depth first search
   3. Alpha-beta pruning
   4. None of the above

**Answer: C** (Alpha-beta pruning)

1. **Which values are independent in the minimax search algorithm?**
   1. Pruned leaves x and y
   2. Every states are dependant
   3. Root is independent
   4. None of the above

**Answer: A** (Pruned leaves x and y)

**Post Lab Subjective Questions:**

* + - 1. **Explain the concept of adversarial search.**

**Ans:**

* Adversarial search , which is most commonly known as Minimax Search ( Or Mini Max Algorithm). This algorithm is mostly used in games to find the best moves. Currently, as a part of AI ( Artificial Intelligence) this algorithm is used in Chess or Tic-Tac-Toe or other Two-Player games.
* It uses the concept of Tree (Data Structure). It navigates through the tree and captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players.
* **Adversarial search** is search when there is an "enemy" or "opponent" changing the state of the problem every step in a direction you do not want.

Examples: Chess, business, trading, war.

You change state, but then you don't control the next state.  
Opponent will change the next state in a way:

* unpredictable
* *hostile*to you
* You only get to change (say) every alternate state.
* One shortcoming of this search is that it assumes that the opponent will act in such a way that he will effectively minimize your gains and maximize his own.
  + - 1. **Explain how alpha-beta pruning improves memory efficiency of algorithm**

**Ans:**

The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated. This way, the search time can be limited to the 'more promising' subtree, and a deeper search can be performed at the same time.

Like its predecessor, it belongs to the branch and bound class of algorithms. The optimization reduces the effective depth to slightly more than half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best choice for side on move ordered first at each node).

With an (average or constant) branching factor of b, and a search depth of d plies, the maximum number of leaf node positions evaluated (when the move ordering is pessimal) is O(b\*b\*...\*b) = O(bd) – the same as a simple minimax search. If the move ordering for the search is optimal (meaning the best moves are always searched first), the number of leaf node positions evaluated is about O(b\*1\*b\*1\*...\*b)

for odd depth and O(b\*1\*b\*1\*...\*1) for even depth, or O ( b d / 2 ) = O ( b d ) { O(b^{d/2})=O({\sqrt {b^{d}}})} O(b^{d/2}) = O(\sqrt{b^d}). In the latter case, where the ply of a search is even, the effective branching factor is reduced to its square root, or, equivalently, the search can go twice as deep with the same amount of computation.[12] The explanation of b\*1\*b\*1\*... is that all the first player's moves must be studied to find the best one, but for each, only the best second player's move is needed to refute all but the first (and best) first player move—alpha–beta ensures no other second player moves need be considered. When nodes are considered in a random order (i.e., the algorithm randomizes), asymptotically, the expected number of nodes evaluated in uniform trees with binary leaf-values is Θ ( ( ( b − 1 + b 2 + 14 b + 1 ) / 4 ) d ) { Θ (((b-1+{\sqrt {b^{2}+14b+1}})/4)^{d})} { Θ (((b-1+{\sqrt {b^{2}+14b+1}})/4)^{d})} .[11] For the same trees, when the values are assigned to the leaf values independently of each other and say zero and one are both equally probable, the expected number of nodes evaluated is Θ ( ( b / 2 ) d ) { Θ ((b/2)^{d})} { Θ ((b/2)^{d})}, which is much smaller than the work done by the randomized algorithm, mentioned above, and is again optimal for such random trees.[9] When the leaf values are chosen independently of each other but from the [ 0 , 1 ] { [0,1]} [0,1] interval uniformly at random, the expected number of nodes evaluated increases to Θ ( b d / log ⁡ ( d ) ) { Θ (b^{d/\log(d)})} { Θ (b^{d/\log(d)})} in the d → ∞ { d\to \infty } { d\to \infty } limit[10], which is again optimal for these kind random trees. Note that the actual work for "small" values of d { d} d is better approximated using 0.925 d 0.747 { 0.925d^{0.747}} { 0.925d^{0.747}}.

* + - 1. **Explain how a game of chess may benefit from the min-max and alpha-beta pruning algorithm.**

**Ans:**

The number of possible ways of playing the first four moves per side in a game of Chess is 318,979,564,000. With such a high branching factor, it becomes impossible to evaluate or even reach the leaf nodes in the state space tree.

The min-max algorithm helps find a solution, but is inefficient and can be optimized for higher efficiency. The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.

The savings of alpha beta can be considerable. If a standard minimax search tree has x nodes, an alpha beta tree in a well-written program can have a node count close to the square-root of x. How many nodes you can actually cut, however, depends on how well ordered your game tree is.

If you always search for the best possible move first, you eliminate the most of the nodes. Of course, we don’t always know what the best move is, or we wouldn’t have to search in the first place.

Conversely, if we always searched for worse moves before the better moves, we wouldn’t be able to cut any part of the tree at all. For this reason, good move ordering is very important, and is the focus of a lot of the effort of writing a good chess program.

**Conclusion**

Thus, we have successfully implemented the Checkers game using the Min-max algorithm.