**19CSE451: Lab Evaluation1**

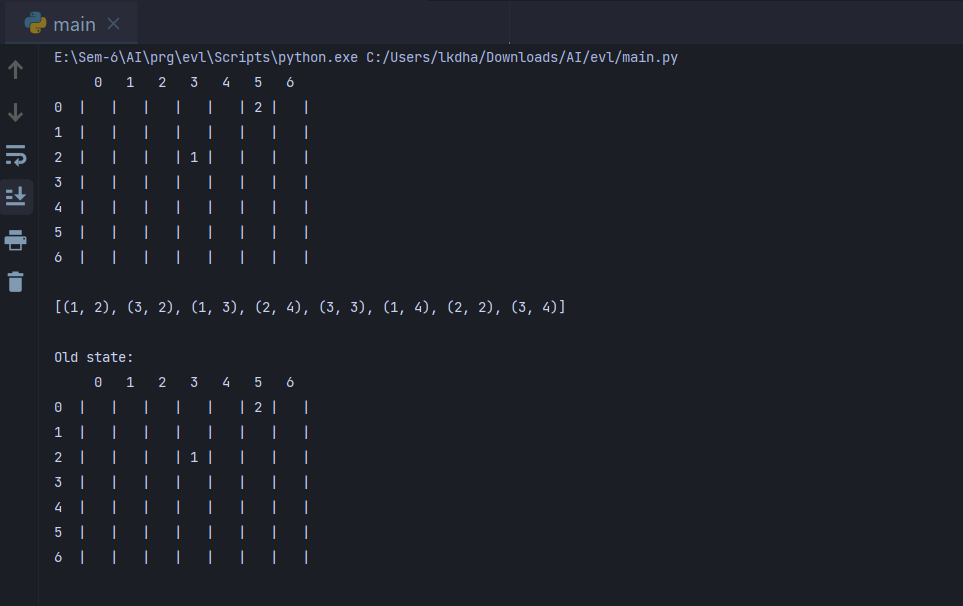
**CB.EN.U4CSE19318 DHARUN NARAYANAN L K**

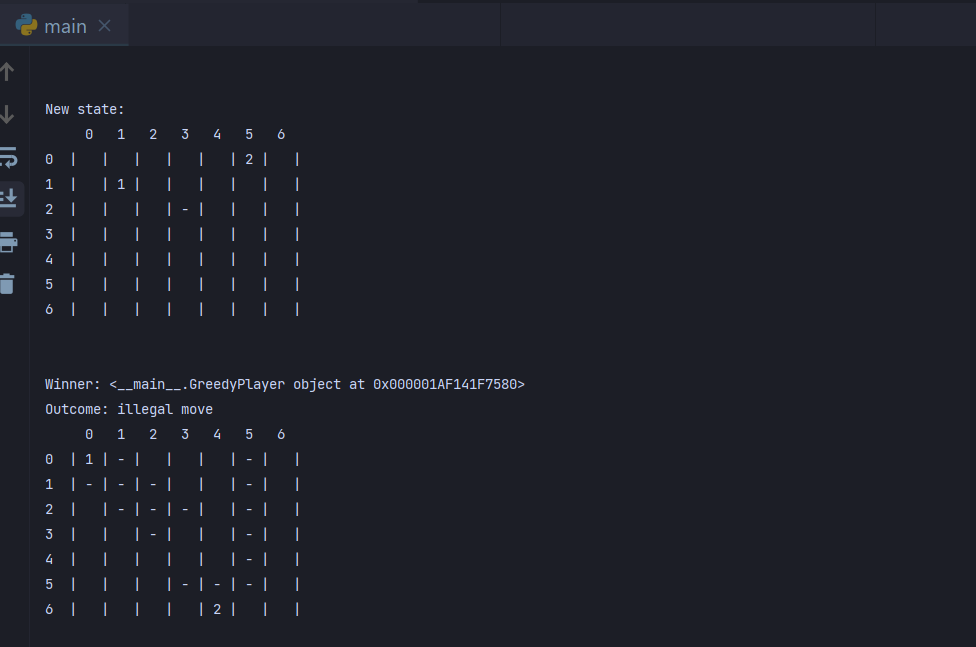
**Isolation(Isola) Using Min-Max Algorithm:**

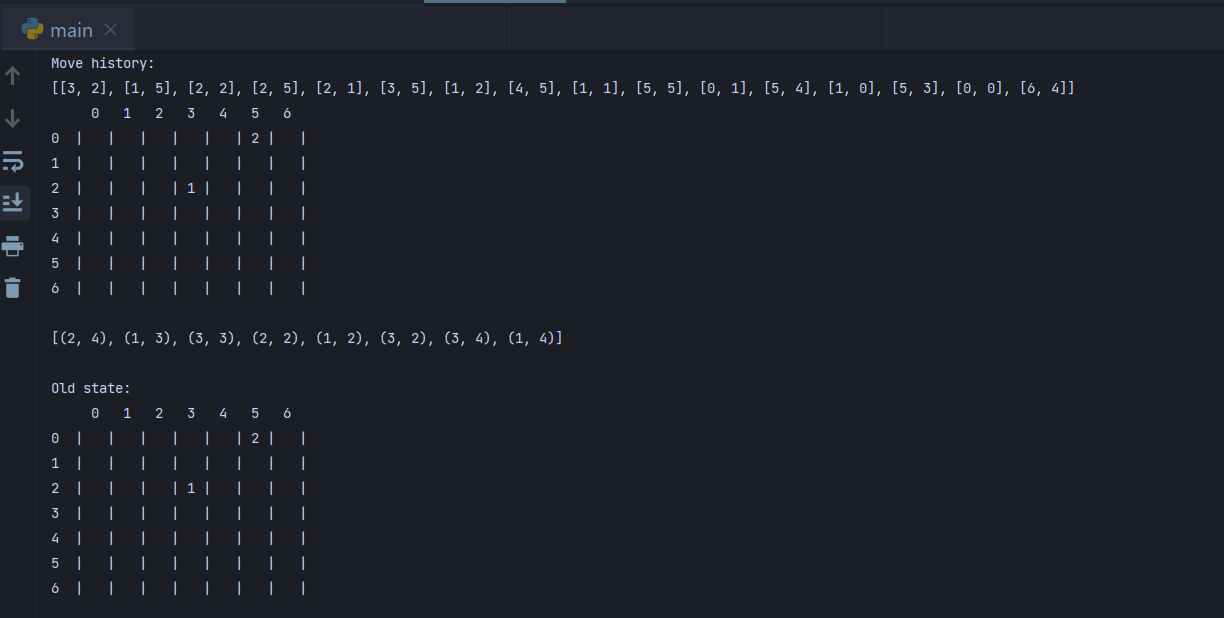
**Code:**

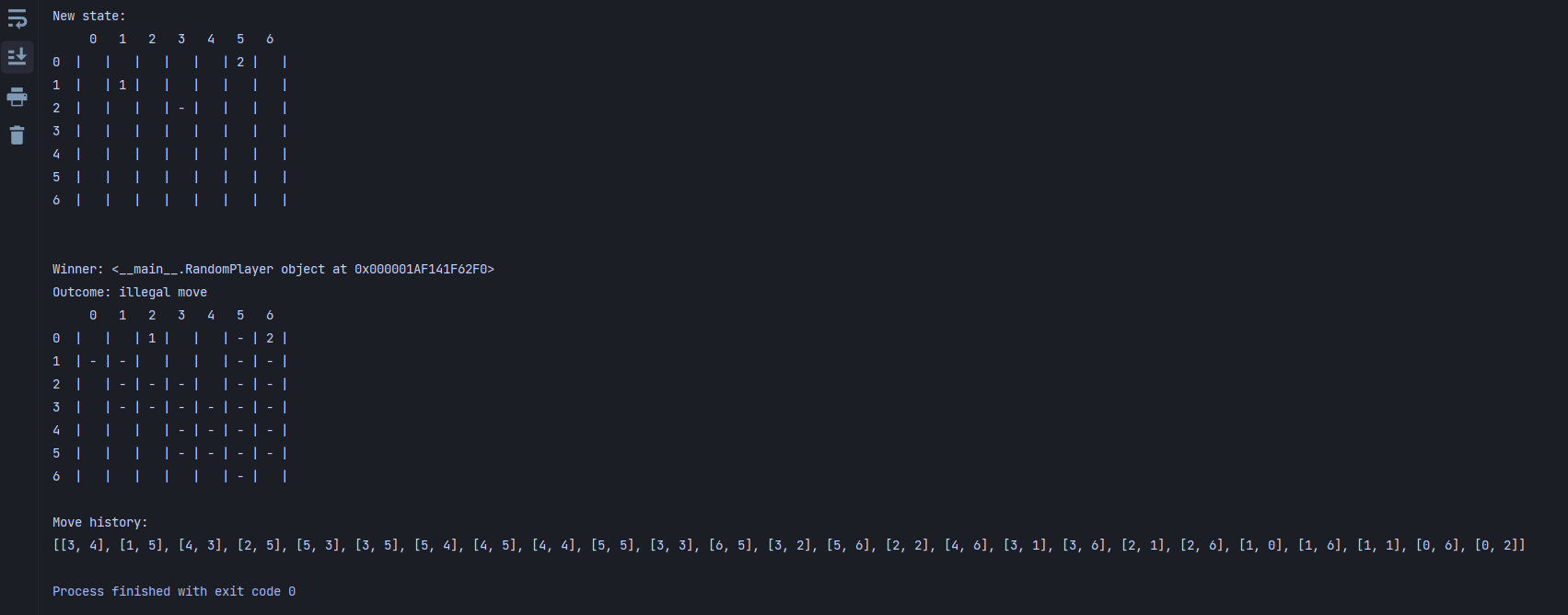
*import* random  
*import* timeit  
*from* random *import* randint  
*from* copy *import* copy  
  
TIME\_LIMIT\_MILLIS = 150  
  
  
*class* Board(*object*):  
 BLANK = 0  
 NOT\_MOVED = *None  
  
 def \_\_init\_\_*(*self*, player\_1, player\_2, width=7, height=7):  
 *self*.width = width  
 *self*.height = height  
 *self*.move\_count = 0  
 *self*.\_player\_1 = player\_1  
 *self*.\_player\_2 = player\_2  
 *self*.\_active\_player = player\_1  
 *self*.\_inactive\_player = player\_2  
  
 *self*.\_board\_state = [Board.BLANK] \* (width \* height + 3)  
 *self*.\_board\_state[-1] = Board.NOT\_MOVED  
 *self*.\_board\_state[-2] = Board.NOT\_MOVED  
  
 *def* hash(*self*):  
 *return str*(*self*.\_board\_state).*\_\_hash\_\_*()  
  
 @property  
 *def* active\_player(*self*):  
 *return self*.\_active\_player  
  
 @property  
 *def* inactive\_player(*self*):  
 *return self*.\_inactive\_player  
  
 *def* get\_opponent(*self*, player):  
 *if* player == *self*.\_active\_player:  
 *return self*.\_inactive\_player  
 *elif* player == *self*.\_inactive\_player:  
 *return self*.\_active\_player  
 *raise RuntimeError*("`player` must be an object registered as a player in the current game.")  
  
 *def* copy(*self*):  
 new\_board = Board(*self*.\_player\_1, *self*.\_player\_2, width=*self*.width, height=*self*.height)  
 new\_board.move\_count = *self*.move\_count  
 new\_board.\_active\_player = *self*.\_active\_player  
 new\_board.\_inactive\_player = *self*.\_inactive\_player  
 new\_board.\_board\_state = copy(*self*.\_board\_state)  
 *return* new\_board  
  
 *def* forecast\_move(*self*, move):  
 new\_board = *self*.copy()  
 new\_board.apply\_move(move)  
 *return* new\_board  
  
 *def* move\_is\_legal(*self*, move):  
 idx = move[0] + move[1] \* *self*.height  
 *return* (0 <= move[0] < *self*.height *and* 0 <= move[1] < *self*.width *and  
 self*.\_board\_state[idx] == Board.BLANK)  
  
 *def* get\_blank\_spaces(*self*):  
 *return* [(i, j) *for* j *in range*(*self*.width) *for* i *in range*(*self*.height)  
 *if self*.\_board\_state[i + j \* *self*.height] == Board.BLANK]  
  
 *def* get\_player\_location(*self*, player):  
 *if* player == *self*.\_player\_1:  
 *if self*.\_board\_state[-1] == Board.NOT\_MOVED:  
 *return* Board.NOT\_MOVED  
 idx = *self*.\_board\_state[-1]  
 *elif* player == *self*.\_player\_2:  
 *if self*.\_board\_state[-2] == Board.NOT\_MOVED:  
 *return* Board.NOT\_MOVED  
 idx = *self*.\_board\_state[-2]  
 *else*:  
 *raise RuntimeError*(  
 "Invalid player in get\_player\_location: {}".format(player))  
 w = idx // *self*.height  
 h = idx % *self*.height  
 *return* (h, w)  
  
 *def* get\_legal\_moves(*self*, player=*None*):  
 *if* player *is None*:  
 player = *self*.active\_player  
 *return self*.\_\_get\_moves(*self*.get\_player\_location(player))  
  
 *def* apply\_move(*self*, move):  
 idx = move[0] + move[1] \* *self*.height  
 last\_move\_idx = *int*(*self*.active\_player == *self*.\_player\_2) + 1  
 *self*.\_board\_state[-last\_move\_idx] = idx  
 *self*.\_board\_state[idx] = 1  
 *self*.\_board\_state[-3] ^= 1  
 *self*.\_active\_player, *self*.\_inactive\_player = *self*.\_inactive\_player, *self*.\_active\_player  
 *self*.move\_count += 1  
  
 *def* is\_winner(*self*, player):  
 *return* player == *self*.\_inactive\_player *and not self*.get\_legal\_moves(*self*.\_active\_player)  
  
 *def* is\_loser(*self*, player):  
 *return* player == *self*.\_active\_player *and not self*.get\_legal\_moves(*self*.\_active\_player)  
  
 *def* utility(*self*, player):  
 *if not self*.get\_legal\_moves(*self*.\_active\_player):  
  
 *if* player == *self*.\_inactive\_player:  
 *return float*("inf")  
  
 *if* player == *self*.\_active\_player:  
 *return float*("-inf")  
  
 *return* 0.  
  
 *def* \_\_get\_moves(*self*, loc):  
 *if* loc == Board.NOT\_MOVED:  
 *return self*.get\_blank\_spaces()  
  
 r, c = loc  
  
 directions = [(0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1)]  
  
 valid\_moves = [(r + dr, c + dc) *for* dr, dc *in* directions  
 *if self*.move\_is\_legal((r + dr, c + dc))]  
 random.shuffle(valid\_moves)  
 *return* valid\_moves  
  
 *def* print\_board(*self*):  
 *return self*.to\_string()  
  
 *def* to\_string(*self*, symbols=*None*):  
 *if* symbols *is None*:  
 symbols = ['1', '2']  
 p1\_loc = *self*.\_board\_state[-1]  
 p2\_loc = *self*.\_board\_state[-2]  
  
 col\_margin = *len*(*str*(*self*.height - 1)) + 1  
 prefix = "{:<" + "{}".format(col\_margin) + "}"  
 offset = " " \* (col\_margin + 3)  
 out = offset + ' '.join(*map*(*str*, *range*(*self*.width))) + '\n\r'  
 *for* i *in range*(*self*.height):  
 out += prefix.format(i) + ' | '  
 *for* j *in range*(*self*.width):  
 idx = i + j \* *self*.height  
 *if not self*.\_board\_state[idx]:  
 out += ' '  
 *elif* p1\_loc == idx:  
 out += symbols[0]  
 *elif* p2\_loc == idx:  
 out += symbols[1]  
 *else*:  
 out += '-'  
 out += ' | '  
 out += '\n\r'  
  
 *return* out  
  
 *def* play(*self*, time\_limit=TIME\_LIMIT\_MILLIS):  
 move\_history = []  
  
 time\_millis = *lambda*: 1000 \* timeit.default\_timer()  
  
 *while True*:  
  
 legal\_player\_moves = *self*.get\_legal\_moves()  
 game\_copy = *self*.copy()  
  
 move\_start = time\_millis()  
 time\_left = *lambda*: time\_limit - (time\_millis() - move\_start)  
 curr\_move = *self*.\_active\_player.get\_move(game\_copy, time\_left)  
 move\_end = time\_left()  
  
 *if* curr\_move *is None*:  
 curr\_move = Board.NOT\_MOVED  
  
 *if* move\_end < 0:  
 *return self*.\_inactive\_player, move\_history, "timeout"  
  
 *if* curr\_move *not in* legal\_player\_moves:  
 *if len*(legal\_player\_moves) > 0:  
 *return self*.\_inactive\_player, move\_history, "forfeit"  
 *return self*.\_inactive\_player, move\_history, "illegal move"  
  
 move\_history.append(*list*(curr\_move))  
  
 *self*.apply\_move(curr\_move)  
  
*def* null\_score(game, player):  
  
 *if* game.is\_loser(player):  
 *return float*("-inf")  
  
 *if* game.is\_winner(player):  
 *return float*("inf")  
  
 *return* 0.  
  
  
*def* open\_move\_score(game, player):  
 *if* game.is\_loser(player):  
 *return float*("-inf")  
  
 *if* game.is\_winner(player):  
 *return float*("inf")  
  
 *return float*(*len*(game.get\_legal\_moves(player)))  
  
  
*def* improved\_score(game, player):  
 *if* game.is\_loser(player):  
 *return float*("-inf")  
  
 *if* game.is\_winner(player):  
 *return float*("inf")  
  
 own\_moves = *len*(game.get\_legal\_moves(player))  
 opp\_moves = *len*(game.get\_legal\_moves(game.get\_opponent(player)))  
 *return float*(own\_moves - opp\_moves)  
  
  
*def* center\_score(game, player):  
 *if* game.is\_loser(player):  
 *return float*("-inf")  
  
 *if* game.is\_winner(player):  
 *return float*("inf")  
  
 w, h = game.width / 2., game.height / 2.  
 y, x = game.get\_player\_location(player)  
 *return float*((h - y) \*\* 2 + (w - x) \*\* 2)  
  
  
*class* RandomPlayer():  
 *def* get\_move(*self*, game, time\_left):  
 legal\_moves = game.get\_legal\_moves()  
 *if not* legal\_moves:  
 *return* (-1, -1)  
 *return* legal\_moves[randint(0, *len*(legal\_moves) - 1)]  
  
  
*class* GreedyPlayer():  
  
 *def \_\_init\_\_*(*self*, score\_fn=open\_move\_score):  
 *self*.score = score\_fn  
  
 *def* get\_move(*self*, game, time\_left):  
 legal\_moves = game.get\_legal\_moves()  
 *if not* legal\_moves:  
 *return* (-1, -1)  
 \_, move = *max*([(*self*.score(game.forecast\_move(m), *self*), m) *for* m *in* legal\_moves])  
 *return* move  
  
  
*class* HumanPlayer():  
  
 *def* get\_move(*self*, game, time\_left):  
  
 legal\_moves = game.get\_legal\_moves()  
 *if not* legal\_moves:  
 *return* (-1, -1)  
  
 *print*(game.to\_string())  
 *print*(('\t'.join(['[%d] %s' % (i, *str*(move)) *for* i, move *in enumerate*(legal\_moves)])))  
  
 valid\_choice = *False  
 while not* valid\_choice:  
 *try*:  
 index = *int*(*input*('Select move index:'))  
 valid\_choice = 0 <= index < *len*(legal\_moves)  
  
 *if not* valid\_choice:  
 *print*('Illegal move! Try again.')  
  
 *except ValueError*:  
 *print*('Invalid index! Try again.')  
  
 *return* legal\_moves[index]  
  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
  
 player1 = RandomPlayer()  
 player2 = GreedyPlayer()  
 game = Board(player1, player2)  
  
 game.apply\_move((2, 3))  
 game.apply\_move((0, 5))  
 *print*(game.to\_string())  
  
 *assert* (player1 == game.active\_player)  
  
 *print*(game.get\_legal\_moves())  
  
 new\_game = game.forecast\_move((1, 1))  
 *assert* (new\_game.to\_string() != game.to\_string())  
 *print*("\nOld state:\n{}".format(game.to\_string()))  
 *print*("\nNew state:\n{}".format(new\_game.to\_string()))  
  
 winner, history, outcome = game.play()  
 *print*("\nWinner: {}\nOutcome: {}".format(winner, outcome))  
 *print*(game.to\_string())  
 *print*("Move history:\n{!s}".format(history))  
  
player1 = RandomPlayer()  
player2 = GreedyPlayer()  
game = Board(player1, player2)  
  
game.apply\_move((2, 3))  
game.apply\_move((0, 5))  
*print*(game.to\_string())  
  
*assert* (player1 == game.active\_player)  
  
*print*(game.get\_legal\_moves())  
  
new\_game = game.forecast\_move((1, 1))  
*assert* (new\_game.to\_string() != game.to\_string())  
*print*("\nOld state:\n{}".format(game.to\_string()))  
*print*("\nNew state:\n{}".format(new\_game.to\_string()))  
  
winner, history, outcome = game.play()  
*print*("\nWinner: {}\nOutcome: {}".format(winner, outcome))  
*print*(game.to\_string())  
*print*("Move history:\n{!s}".format(history))

**Output:**

****

****

****

****