Source Code:

```
##Game: Squeeze IT ##
## Intelligent System Project 1##

import os
import math
from collections import  defaultdict
import copy

## All the Global Variables used in the Program ##
globals()
ls_x=[]
ls_o=[]
ROW_COUNT = 8
COLUMN_COUNT = 8
player1 = {}
player2 = {}
PLAYER = 0
AI = 1
board = [[0 for x in range(ROW_COUNT)] for y in range(COLUMN_COUNT)]
EMPTY = 0
PLAYER_PIECE = "X"
AI_PIECE = "O"
game_name = "Squeeze It"
colors = ["X", "O"]
finished= False
round=0
turn=PLAYER_PIECE
search_current=()
search_choice=()
currrow=0
currcol=0
vr=10
vc=10

## To print the Initial board ##

def print_board():
    print(game_name)
    print("Round: " + str(round))
    for i in range((ROW_COUNT-1), -1, -1):
        print("\t", i, " ", end="")
        for j in range(COLUMN_COUNT):
            print("|", end=" ")
            if (i == 0):
                ls_x.append((i, j))
                print(" "+str(board[i][j])+" ", end=" ")
            elif (i == COLUMN_COUNT-1):
                ls_o.append((i, j))
                print(" "+str(board[i][j])+" ", end=" ")
            else:
                print(" "+str(board[i][j])+" ", end=" ")
        print("| ")
    print("\t\t    0     1     2     3    4     5     6     7")
```

```
## To print the  board After each move ##

def printStateAfter():
    print("Round: " + str(round))
    for i in range(ROW_COUNT-1, -1, -1):
        print("\t", i, " ", end="")
        for j in range(COLUMN_COUNT):
            print("|", end=" ")
            if (i, j) in ls_x:
                board[i][j] = "X"
                print(" "+str(board[i][j])+" ", end=" ")
            elif (i, j) in ls_o:
                board[i][j] = "O"
                print(" "+str(board[i][j])+" ", end=" ")
            else:
                board[i][j] = " "
                print(" " + str(board[i][j]+" "), end=" ")
        print("| ")
    print("\t\t   0     1     2     3    4     5     6     7")

## To Initialize all the players ##

def initilaize_players():
    player1['name'] = None
    player2['name'] = None
    player1['color'] = 'X'
    player1['type'] = 'Human'
    player2['color'] = 'O'
    player2['type'] = 'AI'
    os.system(['clear', 'cls'][os.name == 'nt'])
    print(u"Welcome to {0}!".format(game_name))
    print("Should Player 1 be a Human or a Computer?")
    while player1['name'] == None:
        choice = str(input("Type 'H' or 'C': "))
        if choice == "Human" or choice.lower() == "h":
            player1['name'] = str(input("What is Player 1's name? "))
        elif choice == "Computer" or choice.lower() == "c":
            player2['name']= str(input("What is Player 1's name? "))
        else:
            print("Invalid choice, please try again")

    print("Should Player 2 be a Human or a Computer?")
    while player2['name'] == None:
        choice = str(input("Type 'H' or 'C': "))
        if choice == "Human" or choice.lower() == "h":
            player1['name'] = str(input("What is Player 2's name? "))
        elif choice == "Computer" or choice.lower() == "c":
            player2['name'] = str(input("What is Player 2's name? "))
        else:
            print("Invalid choice, please try again")

    print("{0} will be {1}".format(player1['name'], player1['color']))
    print("{0} will be {1}".format(player2['name'], player2['color']))

    return True

## To create the  board  ##
```

```
def create_board():

    for i in range(ROW_COUNT-1,-1,-1):
        for j in range(COLUMN_COUNT):
            if (i == 0):
                board[i][j]="X"
            elif (i == (ROW_COUNT-1)):
                board[i][j]="O"
            else:
                board[i][j]=" "
    print(board)
    return board

## To drop the  piece at the specified position ##


def drop_piece(board, row, col, piece):
    board[row][col] = piece

## Get pawn locations based on the player ##

def get_pawn_positions(board,player):
    ls=[]
    for i in range(ROW_COUNT):
        for j in range(COLUMN_COUNT):
            if board[i][j]==player:
                ls.append((i,j))
    return ls



## Get valid locations based on the player and the move ##


def get_valid_locations(board,oppo_choice,player):
    if player == AI_PIECE:
        ls=get_pawn_positions(board,AI_PIECE)
    else:
        ls=get_pawn_positions(board, PLAYER_PIECE)
    valid_locations = defaultdict(list)
    oppo_rows=oppo_choice[0]
    oppo_col=oppo_choice[1]

    valid_locations_range = []

    for r in range(oppo_rows + 1, ROW_COUNT):
        if board[r][oppo_col] == " ":
            valid_locations_range.append((r,oppo_col))

    for r in range(oppo_rows - 1, -1,-1):
        if board[r][oppo_col] == " ":
            valid_locations_range.append((r,oppo_col))

    for c in range(oppo_col + 1, COLUMN_COUNT):
        if board[oppo_rows][c] == " ":
            valid_locations_range.append((oppo_rows,c))
    for c in range(oppo_col - 1, -1,-1):
        if board[oppo_rows][c] == " ":
            valid_locations_range.append((oppo_rows,c))


    for (rows, column) in ls:
        for col in range(column + 1, COLUMN_COUNT):
            if board[rows][col] == " " and (rows,col) in
valid_locations_range:
                valid_locations[(rows, column)].append((rows, col))
            elif board[rows][col] == 'X' or board[rows][col] == 'O':
                break

        for col in range(column - 1, -1, -1):
            if board[rows][col] == " " and (rows,col) in
valid_locations_range:
                valid_locations[(rows, column)].append((rows, col))
            elif board[rows][col] == 'X' or board[rows][col] == 'O':
                break

        for row in range(rows + 1, ROW_COUNT):
            if board[row][column] == " " and (row,column) in
valid_locations_range:
                valid_locations[(rows, column)].append((row, column))
            elif board[row][column] == 'X' or board[row][column] == 'O':
                break

        for row in range(rows - 1, -1, -1):
            if board[row][column] == " " and (row,column) in
valid_locations_range:
                valid_locations[(rows, column)].append((row, column))
            elif board[row][column] == 'X' or board[row][column] == 'O':
                break

    return valid_locations

## Switch the player type ##

def switchtype(turn):
    if turn == "AI_PIECE":
        return AI_PIECE
    else:
        return PLAYER_PIECE

# Switch the player turn #

def switchTurn(turn):
    if turn==AI_PIECE:
        turn=PLAYER_PIECE
    else:
        turn=AI_PIECE
    return turn


## Probability Score for the move to win based on the score ##
```

```
def probabilty_score(board,playerturn):
    score=0
    oppo = switchtype(playerturn)
    row = 0
    for col in range(0, COLUMN_COUNT):
        if board[row][col] == playerturn:
            if board[row + 1][col] == oppo:
                score += 6
        elif board[row][col] == oppo:
            if board[row + 1][col] == playerturn:
                score -= 5
    col = 0
    for row in range(0, ROW_COUNT):
        if board[row][col] == playerturn:
            if board[row][col + 1] == oppo:
                score += 6
        elif board[row][col] == oppo:
            if board[row][col + 1] == playerturn:
                score -= 5
    row = ROW_COUNT-1
    for col in range(0, COLUMN_COUNT):
        if board[row][col] == playerturn:
            if board[row - 1][col] == oppo:
                score += 6
        elif board[row][col] == oppo:
            if board[row - 1][col] == playerturn:
                score -= 5

    col = COLUMN_COUNT-1
    for row in range(0, ROW_COUNT):
        if board[row][col] == playerturn:
            if board[row][col - 1] == oppo:
                score += 6
        elif board[row][col] == oppo:
            if board[row][col - 1] == playerturn:
                score -= 5

    for row in range(2,ROW_COUNT-5):
        for col in range(2,COLUMN_COUNT-2):
            if board[row][col]==playerturn:
                if board[row][col+1]==oppo:
                    score+=-2
                elif board[row][col-1]==oppo:
                    score += -2
                elif board[row+1][col + 1] == oppo:
                    score += -2
                elif board[row - 1][col + 1] == oppo:
                    score += -2

            elif board[row][col]==oppo:
                if board[row][col+1]==playerturn:
                    score+= 2
                elif board[row][col-1]==playerturn:
                    score += 2
                elif board[row+1][col + 1] == playerturn:
                    score += 2
                elif board[row - 1][col + 1] == playerturn:
                    score += 2

    return score

## Horizontal moves based the move of the player ##

def horizontalmove(board, playerturn):
    turn = playerturn
    maxval = 0
    group = False
    intermediategrp = False
    notnewend = True
    oppoturn = switchtype(playerturn)
    pairs = []
    finalpair = []
    for r in range(ROW_COUNT):
        maxcount = 0
        for c in range(COLUMN_COUNT):
            if board[r][c] == turn:
                startpos = (r, c)

                for i in range(COLUMN_COUNT - 1, -1, -1):
                    if board[r][i] == turn:
                        if notnewend:
                            endpos = (r, i)
                            group = True
                            notnewend = False

                        elif (r, i) == startpos and not intermediategrp:
                            pairs.append([startpos, endpos, maxcount])
                            group = True
                            if maxcount > maxval:
                                finalpair.append([startpos, endpos])
                                maxval = maxcount
                            break

                        elif (r, i) != startpos :
                            tempstart = startpos
                            startpos = (r, i)
                            pairs.append([startpos, endpos, maxcount])
                            intermediategrp = True
                            if maxcount > maxval:
                                finalpair.append([startpos, endpos])
                                maxval = maxcount
                            endpos = startpos
                            startpos = tempstart
                            maxcount=0

                    elif board[r][i] == ' ':
                        group = False
                        break
                    elif board[r][i] == oppoturn and group:
                        maxcount += 1

                    else:
                        break
    return maxval

## Vertical moves based the move of the player ##

def verticalmove(board, playerturn):
    turn = playerturn
    maxval = 0
    group = False
    intermediategrp = False
    notnewend = True
    oppoturn = switchTurn(playerturn)
    pairs = []
    finalpair = []
    for r in range(ROW_COUNT):
        maxcount = 0
        for c in range(COLUMN_COUNT):
            if board[c][r] == turn:
                startpos = (c, r)

                for i in range(COLUMN_COUNT - 1, -1, -1):
                    # if i - c > 1:
                    if board[i][r] == turn:
                        if notnewend:
                            endpos = (r, i)
                            group = True
                            notnewend = False

                        elif (i, r) == startpos and not intermediategrp:
                            pairs.append([startpos, endpos, maxcount])
                            group = True
                            if maxcount > maxval:
                                finalpair.append([startpos, endpos])
                                maxval = maxcount
                            break

                        elif (i, r) != startpos:
                            tempstart = startpos
                            startpos = (r, i)
                            pairs.append([startpos, endpos, maxcount])
                            intermediategrp = True
                            if maxcount > maxval:
                                finalpair.append([startpos, endpos])
                                maxval = maxcount
                            endpos = startpos
                            startpos = tempstart
                            maxcount = 0

                    elif board[i][r] == ' ':
                        group = False
                        break
                    elif board[i][r] == oppoturn and group:
                        maxcount += 1

                    else:
                        break
    return maxval
```

```
## Diagonal check as the pawn cannot move diagonally ##

def diagonalcheck(search_current, search_choice):
        diagonalcheck = False
        while not diagonalcheck:
            if search_current[0] == search_choice[0] or search_current[1] ==
search_choice[1]:
                return search_choice
            else:
                print(" Cannot make a diagonal move")
                choice_row, choice_col = (input("Enter a move (by row,column
number): ")).split(",")
                search_choice = (int(choice_row), int(choice_col))

## Human Turn ##

def Humanmove(state):
        column = False
        choice = False
        opponent=True
        search_current = ()
        search_choice = ()
        while not column:
            current_row, current_col = (input("Enter a current state (by
row,column number):")).split(",")
            search_current = (int(current_row), int(current_col))
            if search_current in ls_x or search_current in ls_o:
                      while opponent:
                              if (search_current in ls_x or search_current in
ls_o) and search_current in ls_o:
                                    print(" Cannot place at opponents place")
                                    current_row, current_col = (input("Enter
a current state (by row,column number):")).split(",")
                                    search_current = (int(current_row),
int(current_col))
                              else:
                                  opponent=False
                      while not choice:
                            choice_row, choice_col = (input("Enter a move (by
row,column number): ")).split(",")
                            search_choice = (int(choice_row),
int(choice_col))
                            search_choice=diagonalcheck(search_current,
search_choice)
                            if search_choice not in ls_x and search_choice
not in ls_o:
                                ls_x.remove(search_current)
                                ls_x.append(search_choice)
                                choice = True
                            else:
                                print("Cannot make a move.Pawn already exist
in the entered row and value")
                      column = True
            else:
                print("Cannot make a move.Pawn doesn't exist in the entered
row and value")
        return search_current,search_choice
```

```
## Next move to take by Human ##

def nextMove(turn,round):
        search_current, search_choice =Humanmove(turn)
        round+=1
        return round,search_current, search_choice

## Score based on the horizontal and vertical moves ##

def score_position_move(board,currrow,currcol,vr,vc, piece):
      hval=horizontalmove(board,piece)
      vval=verticalmove(board,piece)
      if hval==0 and vval==0:
        return currrow,currcol,vr,vc,probabilty_score(board,piece)
      else:
        return currrow,currcol,vr,vc,max(hval,vval)

## Minimax Algorithm with Alpha-Beta Purning ##

def minimax(board,currrow,currcol,vr,vc,oppo_choice,alpha,beta, depth,
player,maximizingPlayer):
    valid_locations = get_valid_locations(board,oppo_choice,player)
    if depth ==0:
        return score_position_move(board, currrow,currcol,vr,vc, AI_PIECE)
    if maximizingPlayer:   # Maximizing Player
        value = -math.inf
        for (currrow,currcol) in valid_locations.keys():
                visitedo = []
                b_copy = copy.deepcopy(board)
                b_copy[currrow][currcol] = " "
                for vr,vc in valid_locations[(currrow,currcol)]:
                    if(vr,vc) not in visitedo:
                        tempo = copy.deepcopy(b_copy)
                        drop_piece(b_copy, vr,vc, AI_PIECE)
                        oppo_choice=(vr,vc)
                        visitedo.append((vr,vc))
                        currrow,currcol,vr,vc,new_score =
minimax(b_copy,currrow,currcol,vr,vc,oppo_choice,alpha,beta, depth -
1,PLAYER_PIECE, False)
                        b_copy = tempo
                        if new_score > value:
                            vr=vr
                            vc=vc
                            currrow=currrow
                            currcol=currcol
                            value = new_score
                        alpha = max(alpha, value)
                        if alpha >= beta:
                            break
        return  currrow,currcol,vr,vc,value
    else:        # Minimizing player
        value = math.inf
        for (currrow, currcol) in valid_locations.keys():
            visitedx = []
            b_copy = copy.deepcopy(board)
            b_copy[currrow][currcol] = " "
            for vr, vc in valid_locations[(currrow, currcol)]:
                if (vr, vc) not in visitedx:
                    tempx = copy.deepcopy(b_copy)
                    drop_piece(b_copy, vr, vc, PLAYER_PIECE)
                    oppo_choice = (vr, vc)
                    visitedx.append((vr, vc))
                    currrow,currcol,vr,vc,new_score = 
minimax(b_copy,currrow,currcol,vr,vc,oppo_choice,alpha,beta, depth - 1
,AI_PIECE, True)
                    b_copy = tempx
                    if new_score < value:
                        vr = vr
                        vc = vc
                        currrow = currrow
                        currcol = currcol
                        value = new_score
                    beta = min(beta, value)
                    if alpha >= beta:
                        break
        return currrow,currcol,vr,vc,value


initilaize_players()
create_board()
print_board()
game_over = False
exit = False
players = PLAYER_PIECE
search_ai_choice=()

## Code starts her ##

while not exit:
    while not finished:
        if players==PLAYER_PIECE :
            round,search_current,search_choice=nextMove(players,round)
            printStateAfter()
            players=switchTurn(turn)
        else:
            print(" AI turn.......")
            minimax_score = 
minimax(board,currrow,currcol,vr,vc,search_choice,-math.inf, math.inf,
5,players, True)
            search_current1=(minimax_score[0],minimax_score[1])
            search_choice1=(minimax_score[2],minimax_score[3])
            print("Current state: ",search_current1)
            print("To state: ", search_choice1)
            ls_o.remove((search_current1))
            ls_o.append((search_choice1))
            printStateAfter()
            players=switchTurn(AI_PIECE)

        if round > 50:
            finished = True
            if len(ls_x)> len(ls_o):
                print("Human Wins")
            elif len(ls_o)> len(ls_x):
```

```
    print("AI wins")
else:
    print("Game is tie")
print("Game is over")
```