Tetris Played by AI

Ethan DiPilato, Brett Grossman, Abriana Nash

Motivation

- Take already working Tetris source code
- Implement AI to play game using different algorithms
- Baseline: Random approach
- Compare and contrast greedy and optimal approaches
- Tetris uses mathematical strategies; Makes it optimal for using search algorithms

Background

- Played on grid of size 10 cells wide by 22 cells high
- Consists of various shaped blocks that fall at a standard rate
- Speed of movement increases as game progresses
- Goal: arrange pieces at the bottom of grid in horizontal line without gaps so that it disappears
- Loss: when arrangement of pieces surpasses height of grid
- User given advantage by seeing preview of next piece given and allowed to drop piece at faster rate

AI Method

- Design three different agents to play the game (random, “greedy”, “optimal”)
- These agents all “decide” which final positions to play the given piece to
- Final position given to the “solution search” to return a list of actions to be performed on the given playing piece

Solution Search

- Similar to what was done for pacman
- General graph search adapted for the project and application to tetris, implemented as a breadth first search
- Generates a list of left, right, rotate and down actions that will get the piece to the chosen position at the bottom of the board

```python
class SolutionSearch():
    @classmethod
    def isGoalState(self, currentState, goalState):
        return (currentState[0] == goalState[0] and currentState[1] == goalState[1])
    #generates a list of successors of potential states where the stone has moved left/right or rotated
    #state[2] = board   state[5] = stone    state[0] = stone x coordinate   state[1] = stone y coordinate
    @classmethod
    def getSuccessors(self, state):
        successors = []
        if not check_collision(state[2], state[5], (state[0] - 1, state[1])):#moving left/right
            successors.append(((state[0] - 1), state[1],
            newBoard(state[2], state[0], state[1], state[5], state[0] - 1, state[1]), state[3], state[4], state[5], 'LEFT'))
        if not check_collision(state[2], state[5], (state[0] + 1, state[1])):#moving right
            successors.append(((state[0] + 1), state[1],
            newBoard(state[2], state[0], state[1], state[5], state[0] + 1, state[1]), state[3], state[4], state[5], 'RIGHT'))
        if not check_collision(state[2], rotate_clockwise(state[5]), (state[0], state[1])):#rotate stone
            successors.append((state[0], state[1],
            newBoard(state[2], state[0], state[1], rotate_clockwise(state[5]), state[0], state[1]), state[3], state[4], rotate_clockwise(state[5]), 'UP'))
        if not check_collision(state[2], state[5], (state[0], state[1] + 1)):#drop down one line
            successors.append((state[0], state[1] + 1,
            newBoard(state[2], state[0], state[1], state[5], state[0], state[1] + 1)))
        return successors
    @classmethod
    def graphSearch(self, initialState, goalState, frontier):
        explored = []    #list of nodes that have been explored
        frontier.push((initialState, []))   #creates the frontier
        while not frontier.isEmpty(): #continues until the frontier is empty, at the end just returns an empty set in absense of a solution
            state, actions = frontier.pop() #removes from the frontier the current node to be expanded
            if not state in explored: #if that node is not in explored then we expand it and also add it to explored
                explored.append(state)
                if self.isGoalState(state, goalState): #goal state check
                    return actions
                successors = self.getSuccessors(state) #expanding the node
                for successor in successors:    #adding each expansion into the frontier
                    newActions = actions + [successor[6]]
                    frontier.push((successor, newActions))
        return []
```

Final Positions Method

- Loops through the game board searching for and saving all possible final locations for a piece
- Number of loops is determined by the given shape

```python
#generates list of final positions (RETURN LIST OF TUPLES - CONTAINS PIECE'S FINAL X POSITION, FINAL Y POSITION, SHAPE)
def finalPositions(board, piece):
    positions = []
    #if shape is square, loop through finding final positions just once, don't care about rotations
    if piece == tetris_shapes[6]:
        for x in range(10):
            positions.append((x, 0, piece))
            for y in range(22):
                if not check_collision(board, piece, (x,y)):
                    position = positions[x]
                    if y > position[1]:
                        positions.pop()
                        positions.append((x, y, piece))
    #if shape is line, Z or S, loop through twice to take into account a rotation of the piece
    elif piece == tetris_shapes[5] or piece == tetris_shapes[1] or piece == tetris_shapes[2]:
        for z in range(2):
            for x in range(10):
                positions.append((x, 0, piece))
                for y in range(22):
                    if not check_collision(board, piece, (x,y)):
                        position = positions[x]
                        if y > position[1]:
                            positions.pop()
                            positions.append((x, y, piece))
            piece = rotate_clockwise(piece)
    #all other pieces (L/J/T shapes) run four times to take into account the 3 rotations
    elif piece == tetris_shapes[0] or piece == tetris_shapes[3] or piece == tetris_shapes[4]:
        for z in range(4):
            for x in range(10):
                positions.append((x, 0, piece))
                for y in range(22):
                    if not check_collision(board, piece, (x,y)):
                        position = positions[x]
                        if y > position[1]:
                            positions.pop()
                            positions.append((x, y, piece))
            piece = rotate_clockwise(piece)

    return positions
```

Random Agent

- Selects a random move from a list of legal positions for a given piece
- Expected to perform very poorly
- Used as bare minimum baseline for AI's performance

```python
class RandomAgent():
    @staticmethod
    #chooses randomly a final position from a list of all available final positions
    def getAction(gameState):
        actionList = finalPositions(gameState[2], gameState[5])
        print actionList
        return actionList[randint(0,len(actionList)-1)]
```

“Optimal Agent”

- An expectimax agent that evaluates all possible positions for given pieces
- Takes maximum with the current piece, and the previewed piece
- After 2 layers, generates a list of every shape and takes average of best move with each
- Evaluation function is based on score of the game, and max height of the pieces placed

```
class ExpectimaxAgent():

    def getAction(self, gameState):
        return self.value(gameState[2], 3, 0, gameState[5])
    def value(self, board, depthLimit, currentDepth, currentPiece):
        #get successor states if not terminal, returns low negative number if it is terminal.
        if(currentPiece is not 0):
            if(TetrisApp.isGameOver(board,currentPiece)):
                return -9999999
            successorList = list()
            actionList = finalPositions(board, currentPiece)
        #if no shape is set, generate list of shapes and get expected value (average) of best move with each shape
        else:
            avg = 0.0
            for x in tetris.tetris_shapes:
                avg = avg + value(board, depthLimit, currentDepth, x)
            return avg/len(tetris_shapes)
        #return action at top of tree
        if(currentDepth==0):
            scoreHold, action = max([(self.value(newBoard(board,x[2],stone_x,stone_y, x[0],x[1]), depthLimit,1, 1),x) for x in actionList])
            return action
        #for previewed piece
        if(currentDepth==1):
            return max(self.value((newBoard(board,x[2],stone_x,stone_y, x[0],x[1])), depthLimit, 2, 0 )for x in actionList))
        #return score at max depth
        if(currentDepth==depthLimit and currentPiece is not 0):
            return min(evaluationFunction(newBoard(board,x[2],stone_x,stone_y, x[0],x[1])) for x in actionList))
        #all other cases (standard)
        return max(self.value(newBoard(board,x[2],stone_x,stone_y, x[0],x[1])), depthLimit, currentDepth+1, 0) for x in actionList)
    def evaluationFunction(board):
        return board[4] * (1/len(getPieces.asList()))
```

“Greedy” Agent

- Uses the same evaluation function as the expectimax agent, but only in one layer
- Simply evaluates what is the best score you can make with the current piece is
- Poorer game performance, but runs in shorter time and could be better for a very slow machine
- Could be used when the game speeds up to a certain point where it no longer has time to perform the expectimax search

```python
class GreedyAgent():
    def getAction(self, GameState):
        actionList = GameState.finalPositions(board, GameState[5])
        scoreHold, action = max([(evaluationFunction(newBoard(board,x[2],stone_x,stone_y, x[0],x[1])),x)
        for x in actionList])
        return action

    def evaluationFunction(board):
        return board[4] * (1/len(getPieces.asList()))
```

The Road Block

- Struggled with getting the game to perform commands once list of actions was generated
- Difficult to determine how to move the game from taking keyboard inputs to input fed from the action list generated by the agent calls
- Attempted solutions ranged from replacing event polling for keyboard commands with a loop feeding commands out of the actions list to taking the actions list and emulating keystrokes for each command

```python
        while 1:
            self.screen.fill((0,0,0))
            if self.gameover:
                self.center_msg("""Game Over!\nYour score: %d\nPress space to continue""" % self.score)
            else:
                if self.paused:
                    self.center_msg("Paused")
                else:
                    pygame.draw.line(self.screen, (255,255,255), (self.rlim+1, 0), (self.rlim+1, self.height-1))
                    self.disp_msg("Next:", (self.rlim+cell_size, 2))
                    self.disp_msg("Score: %d\n\nLevel: %d\nLines: %d" % (self.score, self.level, self.lines), (self.rlim+cell_size, cell_size*5))
                    self.draw_matrix(self.bground_grid, (0,0))
                    self.draw_matrix(self.board, (0,0))
                    self.draw_matrix(self.stone, (self.stone_x, self.stone_y))
                    self.draw_matrix(self.next_stone, (cols+1,2))
            pygame.display.update()

            state = self.getGameState()
            #this line below runs the Random Agent
            actions = SolutionSearch.graphSearch(state, RandomAgent.getAction(state), Queue())
            #this line below runs the Optimal Agent
            actions = SolutionSearch.graphSearch(state, ExpectimaxAgent.getAction(state), Queue())
            #this line below runs the Greedy Agent
            actions = SolutionSearch.graphSearch(state, GreedyAgent.getAction(state), Queue())

            for event in pygame.event.get():
                if event.type == pygame.USEREVENT+1:
                    self.drop(False)
                elif event.type == pygame.QUIT:
                    self.quit()
                elif event.type == pygame.KEYDOWN:
                    for key in key_actions:
                        if event.key == eval("pygame.K_"+key):
                            key_actions[key]()

            dont_burn_my_cpu.tick(maxfps)
```

Questions?