2048 Development Document

This document explains how an Agent is built for 2048 game. I started with a very basic player and slowly made it smarter. I have documented my thinking and show how well each version worked.

Iteration 1: Implemeted Basic MinMax Agent

The first version of the agent is built using the MinMax algorithm. Using min and max the agent used **iterative deepening**, which means it looks one move ahead until it runs out of time.

Code:

```
from Game2048 import *

class Player(BasePlayer):

    def __init__(self, timeLimit):

        BasePlayer.__init__(self, timeLimit)

    def findMove(self, state):

        actions = self.moveOrder(state)

        depth = 1

        while self.timeRemaining():

            best = -10000

            for a in actions:

                result = state.move(a)

                if not self.timeRemaining(): return

                v = self.minPlayer(result, depth-1)

                if v is None: return

                if v > best:

                    best = v

                    bestMove = a

            self.setMove(bestMove)
        depth += 1
def maxPlayer(self, state, depth):
    if state.gameOver() or depth == 0:
        return self.heuristic(state)
    best = -10000
    for a in state.actions():
        result = state.move(a)
        v = self.minPlayer(result, depth-1)
        if v is None: return None
        if v > best:
            best = v
    return best
def minPlayer(self, state, depth):
    if state.gameOver() or depth == 0:
        return self.heuristic(state)
    best = 1e6
    for (t,v) in state.possibleTiles():
        result = state.addTile(t,v)
        v = self.maxPlayer(result, depth-1)
        if v is None: return None
        if v < best:
            best = v
    return best
def heuristic(self, state):
    return state.getScore()
```

```
        def moveOrder(self, state):

                return state.actions()
```

Results:
It was having scores of
Avg Score: 2,772.54
Maximum Score: 11,364
Which was decent and better than the Random Agents score.

Iteration 2: Greedy Agent

This was a simpler agent that looks only one move ahead. It doesn't try to look into the future, but simply picked the moves that gives the best immediate score.

Code:

```
from Game2048 import *

class Player(BasePlayer):

  def __init__(self, timeLimit):

    BasePlayer.__init__(self, timeLimit)

  def findMove(self, board):

    bestScore = -1000

    bestMove = ''

    for a in board.actions():

      m = board.move(a)

      if m.getScore() > bestScore:

        bestScore = m.getScore()

        bestMove = a

    self.setMove(bestMove)
```

Results:
It gave the scores

Avg Score: 9,869.20
Maximum Score: 15,664
Which did quiet well out performing the Random Agent and Greedy Agent and got a highest score compared to the minmax agent.

Iteration 3: ExpectiMax Agent

In this Iteration we implemented the ExpectiMax algorithm. This is better for a game of chance like 2048. This version is still using the simple score as its heuristic. The ExpectiMax algorithm is performing to maximize the score in specific position in the game board.

Code:

```
from Game2048 import *

class Player(BasePlayer):

  def __init__(self, timeLimit):

    BasePlayer.__init__(self, timeLimit)

  def findMove(self, board):

    moves = board.actions()

    bestMove = moves[0]

    bestScore = -1

    for move in moves:

      nextState = board.move(move)

      score = self.expectimax(nextState, 2)

      if score > bestScore:

        bestScore = score

        bestMove = move

    self.setMove(bestMove)

  def expectimax(self, board, depth):

    if depth == 0 or board.gameOver():
      return board.getScore()

    moves = board.actions()

    total = 0

    for move in moves:

      nextState = board.move(move)

      total += self.expectimax(nextState, depth - 1)

    return total / len(moves) if moves else 0
```

Results:
Avg Score: 9,114.13
Maximum Score: 15,664
Which did quiet well out performing the Random Agent and Greedy Agent and got a highest score compared to the minmax agent.

Iteration 4 : ExpectiMax with a Better Heuristic

This iteration combined the ExpectiMax algorithm with some heuristic functions with implementing empty tiles, max tile with higher-value that is good, monotonicity is implemented with better if the tiles are arranged in increasing order in a pattern and the cornering of the highest tiles.

Code:

```
from Game2048 import *

import random

class Player(BasePlayer):

  def __init__(self, timeLimit):

    BasePlayer.__init__(self, timeLimit)

  def findMove(self, board):

    bestScore = float('-inf')

    bestMove = board.actions()[0]
```

```
        for move in board.actions():
            score = self.expectimax(board.move(move), 4, isMax=False)
            if score > bestScore:
                bestScore = score
                bestMove = move
        self.setMove(bestMove)
    def expectimax(self, board, depth, isMax=True):
        if depth == 0 or board.gameOver():
            return self.heuristic(board)
        if isMax:
            return max(self.expectimax(board.move(move), depth - 1, False) for move in board.actions())
        else:
            total = 0
            possible = board.possibleTiles()
            for (pos, tile) in possible:
                child = board.addTile(pos, tile)
                total += self.expectimax(child, depth - 1, True)
            return total / len(possible) if possible else 0
    def heuristic(self, board):
        score = board.getScore()
        empty = sum(1 for i in range(4) for j in range(4) if board.getTile(i,j)==0)
        maxTile = max(board.getTile(i,j) for i in range(4) for j in range(4))
        monotonicity = 0
        for i in range(4):
            for j in range(3):
```

```
            if board.getTile(i,j) >= board.getTile(i,j+1):
               monotonicity += 1
            if board.getTile(j,i) >= board.getTile(j+1,i):
               monotonicity +=1
      cornerBonus = board.getTile(0,0) * 300
      return score + empty * 150 + maxTile * 300 + monotonicity * 100 + cornerBonus
```

Results:

Avg Score: 16,369.54
Maximum Score: 23,204
Which gave maximum score and highest average score from all the agents implemented till now. Because of the better heuristics added to the expectimax it performed very well and provided a good score.

Iteration 6: Iterative Deepening Expectimax Agent

This agent has given the best results overall which gave maximum score and maximum average score by exceeding all the agents performance. For each possible move, it calls the expectimax function to evaluate the outcome. This function recursively explores future game states, distinguishing between the max node and the expectimax node. The value of any given board is calculated by the heuristic function and through the expectimax node.

Code:

```
from Game2048 import *
import random
import time
class Player(BasePlayer):
   def __init__(self, timeLimit):
      BasePlayer.__init__(self, timeLimit)
   def findMove(self, board):
      bestMove = None
      actions = board.actions()
      if not actions:
          self.setMove(None)
          return
      depth = 1
      while self.timeRemaining():
          currentBestScore = -1e10
          currentBestMove = bestMove if bestMove is not None else actions[0]
          for move in actions:
              if not self.timeRemaining():
                  break
              nextState = board.move(move)
              if nextState._board == board._board:
                  continue
              score = self.expectimax(nextState, depth, False)
              if score > currentBestScore:
                  currentBestScore = score
                  currentBestMove = move
          if self.timeRemaining():
              bestMove = currentBestMove
              self.setMove(bestMove)
          depth += 1

   def expectimax(self, board, depth, isMaxPlayer):
      if depth == 0 or board.gameOver():
          return self.heuristic(board)
      if isMaxPlayer:
          best_val = -1e10
          for move in board.actions():
              nextState = board.move(move)
              best_val = max(best_val, self.expectimax(nextState, depth - 1, False))
          return best_val
      else:
          total_val = 0
          possible_tiles = board.possibleTiles()
          if not possible_tiles:
              return self.heuristic(board)
          for pos, tile_val in possible_tiles:
              child_board = board.addTile(pos, tile_val)
              total_val += self.expectimax(child_board, depth - 1, True)
          return total_val / len(possible_tiles)
   def heuristic(self, board):
      smoothness = 0
      for i in range(4):
          for j in range(3):
              val1_h = board.getTile(i, j)
              val2_h = board.getTile(i, j + 1)
              if val1_h > 0 and val2_h > 0:
                  smoothness -= abs(val1_h - val2_h)
              val1_v = board.getTile(j, i)
              val2_v = board.getTile(j + 1, i)
```

```
                if val1_v > 0 and val2_v > 0:
                    smoothness -= abs(val1_v - val2_v)
        mono_bonus = 0
        for i in range(4):
            row = [board.getTile(i, j) for j in range(4)]
            col = [board.getTile(j, i) for j in range(4)]
            is_mono_row = all(row[k] >= row[k+1] for k in range(3)) or all(row[k] <= row[k+1] for k in
range(3))
            if is_mono_row:
                mono_bonus += 150
            is_mono_col = all(col[k] >= col[k+1] for k in range(3)) or all(col[k] <= col[k+1] for k in
range(3))
            if is_mono_col:
                mono_bonus += 150
        empty_tiles = 0
        for i in range(16):
            if board._board[i] == 0:
                empty_tiles += 1
        max_tile = 0
        for tile in board._board:
            if tile > max_tile:
                max_tile = tile
        corner_bonus = 0
        if board.getTile(0, 0) == max_tile:
            corner_bonus = max_tile * 40000.0
        final_score = (
            smoothness * 1.0 +
            mono_bonus * 1.0 +
            empty_tiles * 270.0 +
            max_tile * 1.0 +
            corner_bonus
        )
        return final_score
```

Results: This iteration has provided the best result score with exceeding all the agents available and provided better than all iterations with an Average score of 27,907.71 and Maximum score of 69,984. Which have been implemented using good heuristics based on board smoothness, tile monotonicity the order, the number of empty spaces, and a large bonus for keeping the maximum tile in a corner, ultimately guiding the agent to make strategically good moves.