Contents

1 Basic Test Results

```
Running Presubmit Test
= = = = = = = = = = start: Q1 = = = = = = = = = =
score: 2608
highest tile: 128
game_duration: 0.3153071403503418
game 1/20 -- Score: (2608, 128)
score: 4688
highest tile: 512
game_duration: 0.41784095764160156
game 2/20 -- Score: (4688, 512)
score: 6856
highest tile: 512
game_duration: 0.6192135810852051
game 3/20 -- Score: (6856, 512)
score: 4208
highest tile: 256
game_duration: 0.45110511779785156
game 4/20 -- Score: (4208, 256)
score: 4164
highest tile: 256
game_duration: 0.4352405071258545
game 5/20 -- Score: (4164, 256)
score: 1692
highest tile: 128
game_duration: 0.214125394821167
game 6/20 -- Score: (1692, 128)
score: 4660
highest tile: 512
game_duration: 0.3977665901184082
game 7/20 -- Score: (4660, 512)
score: 5320
highest tile: 512
game_duration: 0.4632742404937744
game 8/20 -- Score: (5320, 512)
score: 2128
highest tile: 128
game_duration: 0.28502464294433594
game 9/20 -- Score: (2128, 128)
score: 2552
highest tile: 128
game_duration: 0.32010650634765625
game 10/20 -- Score: (2552, 128)
score: 3488
highest tile: 256
game_duration: 0.375917911529541
game 11/20 -- Score: (3488, 256)
score: 8128
highest tile: 512
game_duration: 0.7368602752685547
game 12/20 -- Score: (8128, 512)
score: 3204
highest tile: 256
game_duration: 0.34163880348205566
game 13/20 -- Score: (3204, 256)
score: 3216
highest tile: 256
game_duration: 0.3457508087158203
game 14/20 -- Score: (3216, 256)
score: 3216
```

```
highest tile: 256
game_duration: 0.34131717681884766
game 15/20 -- Score: (3216, 256)
score: 3460
highest tile: 128
game_duration: 0.3855147361755371
game 16/20 -- Score: (3460, 128)
score: 5556
highest tile: 512
game_duration: 0.5053408145904541
game 17/20 -- Score: (5556, 512)
score: 6808
highest tile: 512
game_duration: 0.6266148090362549
game 18/20 -- Score: (6808, 512)
score: 7028
highest tile: 512
game_duration: 0.6508927345275879
game 19/20 -- Score: (7028, 512)
score: 5624
highest tile: 512
game_duration: 0.5081136226654053
game 20/20 -- Score: (5624, 512)
==============================
scores: [2608, 4688, 6856, 4208, 4164, 1692, 4660, 5320, 2128, 2552, 3488, 8128, 3204, 3216, 3216, 3460, 5556, 6808, 7028, 5
highest tile: [128, 512, 512, 256, 256, 128, 512, 512, 128, 128, 256, 512, 256, 256, 256, 128, 512, 512, 512, 512]
game_durations: [0.3153071403503418, 0.41784095764160156, 0.6192135810852051, 0.45110511779785156, 0.4352405071258545, 0.214
win rate: 0.0
= = = = = = = = = = end: Q1 = = = = = = = = = =
```

2 README.txt

```
323379768
206974586
*****
Comments:
Our evaluation function consists of many ideas:
1. Keeping the lower left corner fixed in its place:
It is important to keep a fixed tile in one corner and merge the other tiles
in its direction, so we can keep our solution as clean as possible:
ALWAYS try to move down.
If can't - move left.
If can't - move right.
Only when there's ABSOLUTELY no other option - move up.

2. Maintaining the form of our board as a "snake":
We want our lowest row to have the biggest values, in descending order, and the
rows on top of it need to follow a "snake" pattern, where the tile above the
row can easily be merged with the row when needed.

3. Making sure that there are enough "empty tiles" on the board:
As the game goes on, the amount of empty tiles decreases (sometimes
drastically) which means that the option of maneuvering the board decreases
as well.
We don't want that to happen, so we prioritize boards with bigger chances to
have the "perfect amount" of empty tiles.
DISCLAIMER: PERFECT AMOUNT, IN OUR OPINION, IS 9, SINCE IT MANAGES TO KEEP A
SMALL PYRAMID OF EMPTY TILES IN THE UPPER-RIGHT SIDE OF THE BOARD, WHERE MOST
OF THE MOTION IS USUALLY HAPPENING :)

4. Prioritizing boards with bigger chances to merge with each move:
We calculated the "board's chance to merge" when merging columns (going left
or right) and when merging rows (going up or down) by summing up the
subtractions of each 2 adjacent tiles.
The lower the sum - the better our situation is, since low sums indicate
summing up lots of 0's AKA lots of merges.
We gave the cols a lower weight, so we won't contradict the "snake pattern".

We have tried to play with the weights, make them grant us with higher scores,
but couldn't find a formula that quite suited our expectations :(
The average case is a score of 20,000,
With winning (reaching 2048) in 3/5 cases
```

3 multi agents.py

```python
import math
from collections import namedtuple

import numpy as np
import abc

from numpy import int64

import util
from enum import Enum
from game import Agent, Action
from game_state import GameState
from typing import Tuple


class ReflexAgent(Agent):
    """
    A reflex agent chooses an action at each choice point by examining
    its alternatives via a state evaluation function.

    The code below is provided as a guide.  You are welcome to change
    it in any way you see fit, so long as you don't touch our method
    headers.
    """

    def get_action(self, game_state: GameState):
        """
        You do not need to change this method, but you're welcome to.

        get_action chooses among the best options according to the evaluation function.

        get_action takes a game_state and returns some Action.X for some X in the set {UP, DOWN, LEFT, RIGHT, STOP}
        """

        # Collect legal moves and successor states
        legal_moves = game_state.get_agent_legal_actions()

        # Choose one of the best actions
        scores = [self.evaluation_function(game_state, action) for action in legal_moves]
        best_score = max(scores)
        best_indices = [index for index in range(len(scores)) if scores[index] == best_score]
        chosen_index = np.random.choice(best_indices)  # Pick randomly among the best

        "Add more of your code here if you want to"

        return legal_moves[chosen_index]

    def evaluation_function(self, current_game_state: GameState, action: Action):
        """
        Design a better evaluation function here.

        The evaluation function takes in the current and proposed successor
        GameStates (GameState.py) and returns a number, where higher numbers are better.

        """

        # Useful information you can extract from a GameState (game_state.py)

        successor_game_state = current_game_state.generate_successor(action=action)
```

```
        board = successor_game_state.board
        max_tile = successor_game_state.max_tile
        # score = successor_game_state.score
        score = 0
        if action == Action.UP:
            return 0
        # region Check for descending order in the top left row (with max_tile at the left corner)
        j = board.shape[0] - 1
        prev_tile = board[j][0]
        if board[j][0] == max_tile:
            score += max_tile
            for tile in board[j, 1:]:
                if prev_tile < tile:
                    break
                score += tile
                prev_tile = tile
        score += np.array([(i + 1) * np.sum(board[i, :]) for i in
                           range(board.shape[0])]).sum()  # Pyramid like - give more power for the upper rows

        "*** YOUR CODE HERE ***"
        return score


def score_evaluation_function(current_game_state: GameState):
    """
    This default evaluation function just returns the score of the state.
    The score is the same one displayed in the GUI.

    This evaluation function is meant for use with adversarial search agents
    (not reflex agents).
    """
    return current_game_state.score


class MultiAgentSearchAgent(Agent):
    """
    This class provides some common elements to all of your
    multi-agent searchers.  Any methods defined here will be available
    to the MinmaxAgent, AlphaBetaAgent & ExpectimaxAgent.

    You *do not* need to make any changes here, but you can if you want to
    add functionality to all your adversarial search agents.  Please do not
    remove anything, however.

    Note: this is an abstract class: one that should not be instantiated.  It's
    only partially specified, and designed to be extended.  Agent (game.py)
    is another abstract class.
    """

    def __init__(self, evaluation_function='scoreEvaluationFunction', depth=2):
        self.evaluation_function = util.lookup(evaluation_function, globals())
        self.depth = depth

    @abc.abstractmethod
    def get_action(self, game_state: GameState):
        return


class Agent(Enum):
    Player = 0
    Computer = 1


class MinmaxAgent(MultiAgentSearchAgent):
    def get_action(self, game_state: GameState):
        """
        Returns the minimax action from the current gameState using self.depth
        and self.evaluationFunction.
```

```
128
129         Here are some method calls that might be useful when implementing minimax.
130
131         game_state.get_legal_actions(agent_index):
132             Returns a list of legal actions for an agent
133             agent_index=0 means our agent, the opponent is agent_index=1
134
135         Action.STOP:
136             The stop direction, which is always legal
137
138         game_state.generate_successor(agent_index, action):
139             Returns the successor game state after an agent takes an action
140         """
141         """*** YOUR CODE HERE ***"""
142         minimax = self.minimax(game_state, self.depth, Agent.Player)
143         return minimax[1]
144
145     def minimax(self, game_state: GameState, depth: int, agent: Agent) -> Tuple[int, Action]:
146         # region if     = 0 or v is a terminal node then return  ()
147         if depth == 0 or not game_state.get_legal_actions(0):
148             return self.evaluation_function(game_state), Action.STOP
149         # endregion
150
151         costume_key = lambda x: x[0]
152
153         # region  if isMaxNode then return max
154         if agent == Agent.Player:
155             legal_moves = game_state.get_legal_actions(agent.value)
156             max_val = (float("-inf"), Action.STOP)
157             for move in legal_moves:
158                 new_state = game_state.generate_successor(agent.value, move)
159                 response_val = self.minimax(new_state, depth - 1, Agent.Computer)[0], move
160                 max_val = max(max_val, response_val, key=costume_key)
161             return max_val
162
163         # endregion
164
165         # region  if isMinNode then return min
166         if agent == Agent.Computer:
167             legal_moves = game_state.get_legal_actions(agent.value)
168             min_val = (float("inf"), Action.STOP)
169             for move in legal_moves:
170                 new_state = game_state.generate_successor(agent.value, move)
171                 response_val = self.minimax(new_state, depth, Agent.Player)[0], move
172                 min_val = min(min_val, response_val, key=costume_key)
173             return min_val
174         # endregion
175
176
177 class AlphaBetaAgent(MultiAgentSearchAgent):
178     """
179     Your minimax agent with alpha-beta pruning (question 3)
180     """
181
182     def get_action(self, game_state: GameState):
183         """
184         Returns the minimax action using self.depth and self.evaluationFunction
185         """
186         """*** YOUR CODE HERE ***"""
187         alpha_beta = self.alpha_beta(game_state, Agent.Player, self.depth)
188         return alpha_beta[1]
189
190     def alpha_beta(self, game_state: GameState, agent: Agent, depth: int, alpha=float("-inf"), beta=float("inf")) -> \
191             Tuple[int, Action]:
192         # region End Condition
193         if depth == 0 or not game_state.get_legal_actions(0):
194             return self.evaluation_function(game_state), Action.STOP
195         # endregion
```

```
196
197             costume_key = lambda x: x[0]
198
199             # region alpha pruning
200             if agent == Agent.Player:
201                 legal_moves = game_state.get_legal_actions(agent.value)
202                 return_alpha = (alpha, Action.STOP)
203                 for move in legal_moves:
204                     new_state = game_state.generate_successor(agent.value, move)
205                     alpha = return_alpha[0]
206                     response_val = self.alpha_beta(new_state, Agent.Computer, depth - 1, alpha, beta)[0], move
207                     return_alpha = max(return_alpha, response_val, key=costume_key)
208                     if return_alpha[0] >= beta:
209                         break
210                 return return_alpha
211             # endregion
212
213             # region beta pruning
214             if agent == Agent.Computer:
215                 legal_moves = game_state.get_legal_actions(agent.value)
216                 return_beta = (beta, Action.STOP)
217                 for move in legal_moves:
218                     new_state = game_state.generate_successor(agent.value, move)
219                     beta = return_beta[0]
220                     response_val = self.alpha_beta(new_state, Agent.Player, depth, alpha, beta)[0], move
221                     return_beta = min(return_beta, response_val, key=costume_key)
222                     if alpha >= return_beta[0]:
223                         break
224                 return return_beta
225             # endregion
226
227
228     class ExpectimaxAgent(MultiAgentSearchAgent):
229         """
230         Your expectimax agent (question 4)
231         """
232
233         def get_action(self, game_state: GameState):
234             """
235             Returns the expectimax action using self.depth and self.evaluationFunction
236
237             The opponent should be modeled as choosing uniformly at random from their
238             legal moves.
239             """
240             """*** YOUR CODE HERE ***"""
241             expectimax = self.expctimax(game_state, self.depth, Agent.Player)
242             return expectimax[1]
243
244         def expctimax(self, game_state: GameState, depth: int, agent: Agent):
245             # region End Condition
246             if depth == 0 or not game_state.get_legal_actions(0):
247                 return self.evaluation_function(game_state), Action.STOP
248             # endregion
249
250             costume_key = lambda x: x[0]
251
252             # region Expected Max
253             if agent == Agent.Player:
254                 legal_moves = game_state.get_legal_actions(agent.value)
255                 max_val = (float("-inf"), Action.STOP)
256                 for move in legal_moves:
257                     new_state = game_state.generate_successor(agent.value, move)
258                     response_val = self.expctimax(new_state, depth - 1, Agent.Computer)[0], move
259                     max_val = max(max_val, response_val, key=costume_key)
260                 return max_val
261
262             # endregion
263
```

```
        # region Expected Min
        if agent == Agent.Computer:
            legal_moves = game_state.get_legal_actions(agent.value)
            succesors = []
            for move in legal_moves:
                succesors.append(game_state.generate_successor(agent.value, move))
            succesors = np.array(succesors)
            probability_s = 1 / len(succesors)
            vfunc_expectimax = np.vectorize(self.expctimax)
            responses = vfunc_expectimax(succesors, depth, agent.Player)
            expectation = np.sum(responses[0] * probability_s), Action.STOP
            return expectation

        # endregion
        return


def better_evaluation_function(current_game_state: GameState):
    """
    Your extreme 2048 evaluation function (question 5).

    DESCRIPTION: <write something here so we know what you did>
    Our evaluation function consists of many ideas:
        1. Keeping the lower left corner fixed in its place
        2. Maintaining the form of our board as a "snake"
        3. Making sure that there are enough "empty tiles" on the board
        4. Prioritizing boards with bigger chances to merge with each move
    """
    "*** YOUR CODE HERE ***"
    successor_game_state = current_game_state
    board = successor_game_state.board
    score = np.int64(0)
    max_tile = successor_game_state.max_tile
    features = []
    DEFAULT_WEIGHT = 10
    Feature = namedtuple("Feature", "name weight")
    board_sum = np.sum(board)

    # region Snake Board
    snake = np.array([[1, 2, 4, 8], [128, 64, 32, 16], [256, 512, 1024, 2048], [32768 * 4, 16384 * 4, 8192 * 4, 4096]])
    if max_tile >= 2048:
        snake = np.array([[8, 4, 2, 1], [128, 64, 32, 16], [2048, 1024, 512, 256], [32768 * 4, 16384 * 4, 8192 * 4, 4096]])
    snake = snake / 1024
    score += np.sum(board * snake)
    # endregion

    # region Edge Cases
    moves = current_game_state.get_agent_legal_actions()
    if len(moves) == 1 and moves[0] == Action.UP:
        return float("-inf")
    # endregion

    # region Penalty for max block not being at the corner
    j = board.shape[0] - 1
    if board[j][0] != max_tile:
        features.append(Feature(board[j][0], -10000))
    # endregion

    # region monotonic bottom row
    num_of_descending_bottom_row = 0
    prev_tile = board[j][0]
    for tile in board[j, 1:]:
        if prev_tile < tile:
            break
        num_of_descending_bottom_row += tile
        prev_tile = tile
    features.append(Feature(num_of_descending_bottom_row, 8))
    # endregion
```

```
332
333         # region Empty tiles
334
335         empty_tiles = len(current_game_state.get_empty_tiles()[0])
336
337         features.append(Feature(perfect_num_of_tiles_score(empty_tiles), 10))
338
339         # endregion
340
341         # region pyramid-like rows, best sum row should be at the bottom
342         row_sums = np.sum(board, axis=1)
343         for i, row in enumerate(row_sums):
344             features.append(Feature(row, i + 15))
345         # endregion
346
347         # region merge-ability of a certain board
348         ICKY_VAL = 4496
349         board = current_game_state.board
350         board_dim = board.shape[0]
351         adjacent_cols = np.abs(board[:, :board_dim - 1] - board[:, 1:])  # left to right
352         adjacent_rows = np.abs(board[:board_dim - 1, :] - board[1:, :])  # up to down
353         # the smaller the adjacents_sum - the better our situation
354         # todo: find a way to properly weight this sum.. This seems to work if our
355         #  sum is (1 <= sum <= 4496), it gives a number from 0 to 100, depending on
356         #  how close we are to a good matrix. Not perfect, but good enough
357         adjacent_rows = np.sum(adjacent_rows)
358         adjacent_cols = np.sum(adjacent_cols)
359         if adjacent_rows != 0:
360             ratio = (np.log(adjacent_rows) / np.log(ICKY_VAL))
361             features.append(Feature(ratio, 80))
362         if adjacent_cols != 0:
363             ratio = (np.log(adjacent_cols) / np.log(ICKY_VAL))
364             features.append(Feature(ratio, 25))
365
366         # endregion
367
368         row = board[j, 1:]
369         num_of_merges_max_tile = math.pow(2, math.log(max_tile, 2) - 1) - 1
370         two = np.ones(row.shape).astype(int) * 2
371         if all(row > 0):
372             num_of_merges_row = np.sum(np.power(two, np.log2(row) - 1) - 1)
373             if (num_of_merges_max_tile - num_of_merges_row) > 0:
374                 new = 1 / math.ceil(num_of_merges_max_tile - num_of_merges_row)
375                 features.append(Feature(new, 7))
376
377         for feature in features:
378             i = np.int64(1)
379             i *= feature.name
380             i *= feature.weight
381             i *= board_sum
382             score += i
383         return score
384
385
386     def perfect_num_of_tiles_score(real_amount, min_amount=1, best_amount=9):
387         ratio = 100 / (best_amount - min_amount)
388         if (real_amount <= best_amount):
389             return ratio * (real_amount - 1)
390         else:
391             rest = real_amount - best_amount
392             return 100 - (ratio / 2) * rest
393
394
395     # Abbreviation
396     better = better_evaluation_function
```