2048 Game Using Expectimax

Project Members

- ✓ Syed Abdul Wahab (P15-6077)
- ✓ Muhammad Bilal (P15-6068)
- ✓ Hamza Javed Durrani (P15-6027)

Introduction

2048 is a sliding tile style game played on a four-by-four grid. It is similar in style to rush-hour and the 15 puzzle, but is a very different type of game. It is not a puzzle, in the traditional sense, as you cannot look at the board and calculate the optimal path to the goal. The stochastic nature of the game requires one to develop a strategy for maximizing the score.

Play Online: 2048

Goal of the game

2048 is played on a 4 X 4 grid, with four possible moves up, left, down and right. The objective of the game is to slide numbered tiles on the grid to combine them and create a tile with the number 2048. There are four possible moves from which a player can choose at each turn. After each move, a new tile is inserted into the grid at a random empty position. The value of the tile can be either 2 or 4.

How to run the project

- ✓ Open the **index.html** file in your browser
- ✓ Project will run automatically.

AI Algorithm

We forked gabrielecirulli's 2048 repository and wrote the AI on top of it. This saved a lot of time, as we just had to write the AI, and not implement the whole game from scratch. The main file that we made is **artificialIntelligence.js**. We also made changes in **application.js** to give best move to game once it is calculated by AI algorithm. For Score calculation we made changes in **grid.js** and modify the actual score calculation function.

We used a depth bounded expectimax search to write the AI. At every turn, the AI explores all the four possible directions and then decides the best one. The recurrence relation that it follows is

$$Move(state, depth, agent) = \left(\begin{array}{cc} Score(state) & \text{if } depth = 0 \\ Max_{dir \in directions(state)}(Move(Child(state, dir), depth - 1, GRID)) & \text{if } agent = PLAYER \\ \sum_{tile \in emptyTiles(state)} Prob(state, tile) * Move(inserTile(state, tile), depth - 1, PLAYER) & \text{if } agent = GRID \end{array} \right)$$

The max node is the one in which the player chooses a move (out of the four directions), and the chance node is the one in which the board inserts a random tile. The max node decides the correct move by maximizing the score of its children, which are the chance nodes. Due to space and time restrictions, it is obviously not possible to explore the complete search space. The search is bounded by a depth, at which the AI evaluates the score of the state (the leaf node) using some heuristic.

The pseudo code is given below.

```
bestMove(grid, depth, agent):
  if depth is 0:
    return score(grid)
  elif agent is BOARD:
    score = 0
    for tile in grid.emptyTiles():
      newGrid = grid.clone()
      newGrid.insert(tile, 2)
      score += 0.9 * bestMove(grid, depth - 1, PLAYER)
      newGrid = grid.clone()
      newGrid.insert(tile, 4)
      score += 0.1 * bestMove(grid, depth - 1, PLAYER)
    return score/len(grid.emptyTiles())
  elif agent is PLAYER:
    score = 0
    for dir in [left, up, right, down]:
      newGrid = grid.clone()
```

```
            newGrid.move(dir)

            score = max(score, bestMove(newGrid, depth - 1, BOARD)

        return score
```

- **Use of Heuristics**

  We used a combination of two heuristics to evaluate how **good** a state is.

  ✓ **Using a weight matrix**

  In most of the games that we came close to winning, the bigger tiles were around the corner. Hence the first idea that we used was to push the higher value tiles to one corner of the grid. For this, we assigned different weights to different cells. The score of a given 4 X 4 grid was calculated as following

```
score = 0

for i in range(0, 4):

    for j in range(0, 4):

        score += W[i][j] * grid[i][j].value
```

  This had another advantage, the new randomly generated tiles got generated in the opposite corner, and hence were closer to the smaller value ones. This increased the chances of the newly generated tiles getting merged.

  - **Forming clusters of equal valued tiles**

  It is obviously better for us if more tiles get merged. This happens if in a state two same valued tiles are present next to each other. We calculate another value, a penalty, and subtract it from the score calculated from heuristic one. This penalty is calculated as following:

```
penalty = 0

for each tile:

    for each neighbour of tile:

        penalty += absolute(tile.value - neighbour.value)
```

This penalty becomes large when high value tiles are scattered across the grid, hence indicating that that particular state is bad. The heuristic function I'm using calculates returns **score-plenty**, each calculated as mentioned above.

- **Optimizations of search**

  As far as we know, there is no method to prune an expectimax search. The only way out is to avoid exploring branches that are highly improbable, however this has no use for us as each branch has an equal probability (the new tile has equal probability of popping up on any of the empty tiles).

  Initially we was exploring nodes even if the move played by the PLAYER had no effect on the grid (i.e. the grid was stuck and could not move in a particular direction, which is a common occurrence during gameplay). Eliminating such branches did enhance the performance a bit.

- **Performance of the AI**

  The solver performs quite well. With a search depth of 6, it formed the 2048 tile 9 times out of 10. With a search depth of 8, it formed the 2048 tile every time we tested it, and went on to 4096 every time as well.

  For a better winning rate, and taking the time taken to search per move into consideration, we keep a 8 play look ahead if the number of empty tiles in the grid is less than 4, otherwise a 6 ply look ahead. This combination leads to a win every time, and 8 out of 10 times forms the 4096 tile.

  The AI achieved a best score of 177352, reaching the 8192 tile. The average score was around 40000, and the AI always won the game.