Solving 2048

Vivek Sah

March 9, 2016

1 Introduction

In this paper, we are going to discuss implementation of the game 2048 and an AI to solve it. 2048 is a single-player puzzle game created in March 2014 by 19-year-old Italian web developer Gabriele Cirulli, in which the objective is to slide numbered tiles on a grid to combine them and create a tile with the number 2048. We will discuss the gameplay in detail below, and follow that with discussion of how the game was implemented in this project. Finally, we will discuss the alpha-beta algorithm and the heuristic we used to solve the problem.

2 Gameplay

The puzzle is set on a 4 X 4 grid whose each cell can be slided left, right, down or up. At first, two random blocks are populated with either 2 or 4. After that, the player makes a move(left, right, down or up). If the player made a 'left' move, all the populated blocks(blocks with value) will slide to the left until they hit another block of same value, another block of different value or reach the left-most edge of the grid. In the case, they hit a block with identical value, the two blocks merge ,and the updated value of the merged block will be twice of its previous value. The same logic applies to all the other moves. After each move, one out of the remaining empty cells will be randomly populated with either 2 or 4. If in the course of game, the player cannot make any move, the player loses. On the other hand, if the player succeeds in creating a block with value '2048', the player wins. In this game, we also keep track of scores, whenever two blocks merge, the player's score increases by twice the value of the cells which merged. For example, if the score was 230, and the player creates a new merged block from 64 and 64, the player's score increases by 128.

3 Implementation of 2048

The game itself is implemented in*Board2048.java*. The *Board2048* class initializes the grid and applies the moves given by the user. The 4 X 4 puzzle grid is stored in an array of size 16. Each index in the array points to a Cell class. The cell class stores the x and y coordinate, its index in the array and the value of the cell. It has getter and setter functions for those fields. The most commonly used functions are: *addToGrid()* and *apply(move)*. The *addToGrid()* function populates a random empty cell with either 2 (with a probability of 0.9) or 4 (with a probability of 0.1). This method is called after every move. Below is the code snippet for addToGrid

Listing 1: addTogrid *Board*2048.*java*

```java
public void addToGrid(){
    ArrayList<Cell> emptycells = getEmptyCells();
    if(emptycells.size() == 0){
        playerLoses = true;
        return;
    }
    int i = random.nextInt(emptycells.size());
    int k = (random.nextFloat() < 0.9) ? 2 : 4;
    emptycells.get(i).setValue(k);
}
```

The *apply(move)* method is a bit complicated. We will talk about applying 'left' move as it has the same logic as other moves. To apply 'left' move, we will pick the cells which lie in the second column from the left. For each cell, we will do the following: search to the left until we hit a wall or a cell with non-zero value. If we hit a wall, we update the value of the cell near the wall(if it is zero) to the value of the current cell, and set current cell equal to 0. If we find a non-zero cell(say, C2) to the left, we will update the value of the cell to the immediate right of C2 with the value of current cell. If the cell has the same value, we will double its value and clear the current cell. We will repeat this process for each cell in the the remaining two columns (the third column from the left and the rightmost column from the left). Let's look at the code snippet for 'left' move.

Listing 2: goLeft() *Board*2048.*java*

```
tempInd1[] = { 1,5,9,13};//second leftmost cells(second column
   from the left)
(int i = 0; i < 4 ; i++) { //loop through each cell in that
   column
for (int k = 0; k <3 ; k++) { //loop through each cell to the
   right of this cell
    int p = 1;
    //increase p until we hit the wall or find a non-empty cell
    while((tempInd1[i] + k - p) > (tempInd1[i]-1) &&
        board[tempInd1[i] + k - p].getValue() == 0 ){
        p++;
    }
    if (board[tempInd1[i] + k - p].getValue() ==
        board[tempInd1[i] + k].getValue() &&
        !(board[tempInd1[i] + k - p].merged)) { //merge
        int newValue = board[tempInd1[i] + k - p].getValue() +
            board[tempInd1[i] + k].getValue();
        if(board[tempInd1[i] + k - p].getValue() ==
            board[tempInd1[i] + k].getValue()){
            score+= newValue;
        }
        board[tempInd1[i] + k - p].setValue(newValue);
            //update the value
        board[tempInd1[i] + k].setValue(0);
        numMoved++;
        board[tempInd1[i] + k - p].merged = true; //set that
            cell to mergedStateus so that no other cell can
            merge
        //to this cell in the same move
        mergedcells.add((tempInd1[i] + k - p));

        if (newValue == 2048){
            playerWins = true;
        }
```

```java
    //if we hit the wall, drag the current cell all the way to the left
    } else if(board[tempInd1[i] + k - p].getValue() == 0) {
        board[tempInd1[i] + k - p].setValue(board[tempInd1[i] + k].getValue());
        board[tempInd1[i] + k].setValue(0);
        numMoved++;
    } else{
        if(p!= 1){
            board[tempInd1[i] + k - p+1].setValue(board[tempInd1[i] + k].getValue());

            board[tempInd1[i] + k].setValue(0);
            numMoved++;
        }

    }

}
```

4 Alpha-Beta AI

We can assume that 2048 is a zero sum game where the opposing player is the computer. The player will try to make the moves which maximize his/her chances of winning, while the computer can fill up the cells in a way that the player has no moves to make (however, in reality, the computer is randomly filling the empty cells). In this case, we can use mini-max algorithm with alpha-beta pruning. Alpha-beta pruning allows us to make faster decisions for a given depth. The implementation follows the standard implementation as described in *'Artificial Intelligence: A Modern Approach'* by Russel and Norvig. The only change which needs to be described is the *maxValue* method which describes the moves made by the computer. As we tried all the possible moves for a player, the possible moves for computer, in this case, are filling up each remaining empty space with 2 or 4 in each iteration.

Listing 3: snippet of maxValue *AlphaBetaAI.java*

```java
List<Board2048.Cell> emptycells = board.getEmptyCells();//get
    empty cells
int[] possibleValues = {2, 4}; //possible values to pupulate
    the cells with
mainloop:
for (Board2048.Cell emptyCell : emptycells) {
    for (int v : possibleValues) {
        Board2048 newboard = new Board2048(board); //make new
            board
        newboard.getCell(emptyCell.getBoardIndex()).setValue(v);//set
            one of each possible values to each cell
        int currentValue = (Integer) minValue(newboard, alpha,
            beta, currentDepth + 1, maxDepth).get("value");
        if (currentValue < beta) { //minimize best score
            beta = currentValue;
        }
        if (beta <= alpha) {
            break mainloop; //alpha cutoff
        }
    }
}
```

5 Evaluation Function

The most straightforward way to evaluate the game would be to evaluate the player's score and make moves which maximize the score for the player. However, it is not very effective. For example, some moves might increase a player's score in the short run but might lead to a state where there are cells with different values scattered throughout the board, which will end up in player's loss. So, we need to consider additional factors such as giving more priority to freeing space when empty cells are scarce and keeping larger-valued cells close to each other (such as keeping 8 close to 16, 4 close to 8 and so on). Let us talk about the first additional heuristic component: prioritizing empty cells. We want to give more priority to the number of empty cells when they are scarce and less importance to them when there are plenty. More naturally, when the score is low, the empty cells are less important and vice versa. One way to do it, as suggested in an article on *datumbox.com*, is to take weight the number of empty cells by the logarithm of the score.

Now, let us go over how to keep the larger values together so that they are easier to combine. In the same article, there is a very good explanation on how to achieve this. The writer suggests that we calculate a 'clustering score' for a populated cell and adding all of them to get a clustering score for the board. To do this, we basically, take the average of the scores of a cell and each of its neighboring cell. If the difference in value between neighboring cells are higher, the overall clustering score will be low and vice versa. Combining all these factors, we get a formula for the heuristic score of the board:

$$heuristicScore = score + log(score) * numEmptyCells + clusteringScore$$

The code snippet for clustering score is given below(is a modification of original code by *Vasilis Vryniotis*) :

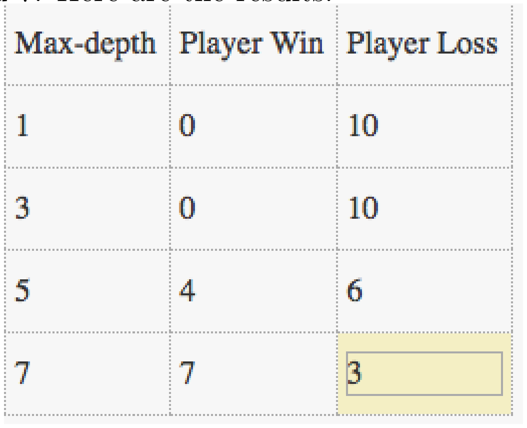
Listing 4: calculateClustering *AlphaBetaAI.java*

```java
private int calculateClustering(Board2048 board){
    int possibleNeighborIndices[] = {-1, 1,-4, 4};
    int clusteringValue = 0;
    for (int i = 0; i < 16; i++) {
        int numberOfNeighbors = 0;
        int sumScoreOfNeighbors = 0;
        for(int n: possibleNeighborIndices){
            if(i+n < 0 || i+n> 15){
                continue;
            }
            if(board.getCell(i).getValue() == 0){
                continue;
            }
            if(board.getCell(i).getValue()>0) {
                numberOfNeighbors++;
                sumScoreOfNeighbors+=Math.abs(board.getCell(i).getValue()-board.get
            }
        }
        if(numberOfNeighbors >= 1)
```

```
            clusteringValue+=sumScoreOfNeighbors/numberOfNeighbors;
    }
    return clusteringValue;
}
```

6 Results

Due to the randomized nature of the game, the results are not consistent throughout many trials. However, 10 trials were conducted for level 1, 3. 5 and 7. Here are the results:



Max-depth	Player Win	Player Loss
1	0	10
3	0	10
5	4	6
7	7	3

Acknowledgment:

Machine Learning Blog & Software Development News. (n.d.). Retrieved March 08, 2016, from http://blog.datumbox.com/using-artificial-intelligence-to-solve-the-2048-game-java-code/