Final Project

Alma Wang

March 9, 2016

1 Introduction- 2048

2048 is a popular puzzle game that involves a 4 x 4 grid of with randomly placed tiles, each containing a value that is an exponent of two. Every round a new tile appears with a value of either 2 or 4 where the probability of 2 is .9 and the probability of a 4 is .1. The player can move all the tiles either left, right, up or down. If the tiles move, then a new tile is added. The score is the sum of new tile values whenever a tile merges. The starting goal of the game is to obtain a tile with the value 2048, but variations involve obtaining the largest tile value possible at the end of the game when all spaces are filled. 2048 is interesting because it is not a zero sum game, but the unpredictably in the new tile and value that appears prohibits from using a straightforward search; it is not a solvable game. Therefore an intelligent solver for the game requires accounting for differences in tile placement.

The game was popularized by Gabriele Circulli and subsequent mobile app versions of the game. Unlike other games, the "creator" of 2048 maintains the availability of the code online enabling others to create many variations online. In 2015, the 2048 game appeared as a competition in the 18th Computer Olympiad. Competitors created solvers for the game and competed for maximum score, average score, maximum tile, and the percentage of times the solver is able to obtain the tile 2048, 4096, 8192 and so on. A Taiwan 2048 Bot tournament also took place in 2014. The winners of the Taiwan tournament implemented variations of the expectedMax algorithm. Other solving algorithms include a Random AI and Alpha Beta.

2 Expected Max AI

Since the 2048 game is non adversarial in nature, it is not an ideal fit for the MiniMax or Alpha Beta pruning algorithms. 2048 aims to maximize its score against a random tile placements. The probability of a particular tile placement is predictable: 1/(number of available spaces) times .1 for tile value of 4 or .9 for a tile value of 2. Therefore the expected utility of a move can be calculated by multiplying the maximum utility of a state my its probability of occurring. This occurs in `ExpectedMax.java`

The implementation of the expected-max algorithm takes place in two parts: maximizing utility of the moves (`maxUtility()`) and calculating the expected utility based on the probability a particular tile appears (`newPieceUtility()`). The `maxUtility()` method tests each move and if the move causes a change in the board, `game.makeMove()` returns true so the utility of the new move is calculated using `newPieceUtility()`. The recursion ends when the game is over or the depth reaches the maximum depth. The `newPieceUtility()` method takes every possible tile placement and calculates the new maximum utility of the new state and weights it by the probability of the tile placement actually occuring. The code for both are shown below:

```
private int maxUtility(Game2048 game, int depth, int numFours){
  if(transpositionTable.containsKey(game.hashCode()))
    return transpositionTable.get(game.hashCode());
  if(cutoffTest(game,depth)){
    int util=utility(game);
    transpositionTable.put(game.hashCode(), util);
    return util;
  }
  List<Integer> moves=game.getMoves();
  int bestUtility=0;
  for(int move:moves){
    if(game.makeMove(move)){
      int util=0;
      util=newPieceUtility(game,depth+1,numFours);
      if(util>bestUtility)
        bestUtility=util;
    }
    game.undoMove();
  }
  return bestUtility;
}
private int newPieceUtility(Game2048 game, int depth, int numFours) {
  double sum=0;
  List<Integer> availableTiles=game.availableSpace();
  for(Integer t:availableTiles){
    int util;
    if(Math.pow(.1, numFours+1)*Math.pow(.9, depth-numFours-1)>PROBCONSTANT){
      // Tile with value 4
      game.addTile(t, 4);
      util=maxUtility(game,depth,numFours+1);
      sum+=util*.10000;
      game.removeTile(t);
      // Tile with value 2
      game.addTile(t, 2);
      util=maxUtility(game,depth,numFours);
      sum+=util*.90000;
      game.removeTile(t);
    }else{
      // Tile with value 2
      game.addTile(t, 2);
      util=maxUtility(game,depth,numFours);
      sum+=util;
      game.removeTile(t);
    }
  }
  return (int)(sum/availableTiles.size());
}
```

The major drawback of the algorithm branching by 2*Number of available space in order to calculate the expected utility. I implemented a transposition table and pruning by means of setting a limit for the probability of sequence of tiles appears. The transposition table is implemented as a HashMap that maps the hashCode of the state of a game to its utility. Before calculating a particular tile of value 4 placement, the method checks that the probability of the values occurring is greater than the PROBCONSTANT, currently set at .00001 or 10^{-5}in order to prune the search.

3 Alpha Beta AI

In order to implement alpha beta pruning, there must be an adversarial opponent. In this case the tile placement could be viewed as the adversary that attempts to place the worst possible tile value and location within reason. Therefore the alpha beta algorithm improves performance in the worst case scenario. This is implemented by changing `newPieceUtility()` to a method that attempts to minimize the utility, as shown in the excerpt below. The process for a tile value of 4 is the same as the one for a tile value of 2 and therefore the code is omitted:

```
private int newPieceUtility(Game2048 game, int depth, int numFours, int alpha, int beta) {
    int minUtility=Integer.MAX_VALUE;
    List<Integer> availableTiles=game.availableSpace();
    for(Integer t:availableTiles){
      // Tile with value 2
      game.addTile(t, 2);
      int utility;
      if(transpositionTable.containsKey(game.hashCode()))
        utility=transpositionTable.get(game.hashCode());
      else
        utility=maxUtility(game,depth,numFours,alpha,beta);
      if(utility<minUtility)
        minUtility=utility;
      if(minUtility<=alpha){
        game.removeTile(t);
        return minUtility;
      }
      if(minUtility<=beta)
        beta=minUtility;
      game.removeTile(t);
            if(!NOFOURS){
              if(Math.pow(.10000000, numFours+1)*Math.pow(.9000000, depth-numFours)>
    PROBCONSTANT){
                  // Tile with value 4
                  game.addTile(t, 4);
                  if(transpositionTable.containsKey(game.hashCode()))
                      utility=transpositionTable.get(game.hashCode());
                  else ...
```

By being able to prune, the alpha beta AI is able to search at a deeper level than the expected max algorithm in a shorter period of time, but the assumption of the worse tile placements is not accurate. With alpha beta, the player assumes that the tiles will always have the value of four, which adversely affects move selection. The PROBCONSTANT feature combats this at higher levels of search, but for lower levels it is ineffective. Therefore I added a feature NOFOURS that assumes only tiles with a value of 2 are being placed. This significantly speeds up performance since only half the tile placements are checked.

4 Random AI

One interesting proposed by users online is to simply take the average utility of a move by making the move and performing a series of random runs that continue until the game ends. This eliminates the branching problem associated with predicting tile placements and therefore performs much faster. Additionally, by being able to perform a high level or runs quickly, the resulting average utility for a move is relatively accurate. This is implemented in `RandomPlayer.java` where the method `randomUtility()` recursively picks a random move and randomly places a new tile on the board until the cutoff test returns true (when the player loses or achieves the target score). An excerpt of the moves selection code is provided below:

```
int bestUtility=0;
for(int move:moves){
  int sum=0;
  for(int i=0;i<runs;i++){
    Game2048 newGame=(new Game2048(game));
    if(newGame.makeMove(move)){
      newGame.addTile();
      sum+=randomUtility(newGame);
    }
  }
int util=sum/runs;
  if(util>=bestUtility){
    bestUtility=util;
    bestMove=move;
  }
}
```

This AI performs faster and generally better than the other AIs due to the high depth and lack of branching. It was able to achieve 2048 more consistently than low level searches with the other AIs.

5 Heuristics

Once the maxDepth of the is reached in the expected max and alpha beta players, the utility of the current state of the board is evaluated. The simplest means of evaluating the state to take the score at the state since the score accounts for tiles merging and depth of the game, but this proves ineffective when scores do not change and/or a particular move that increases the score makes it more difficult to achieve the target tile. Therefore I implemented a number of additional heuristics to improve evaluating utility:

5.1 BLANKSPACES

This heuristic awards bonuses to states with more available spaces. The rationale is that having more blank spaces prevents the player from reaching a losing position where all spaces are filled and not moves are possible. The implementation simply involves adding 10 to the total score for the number of spaces available in a state

5.2 EDGES

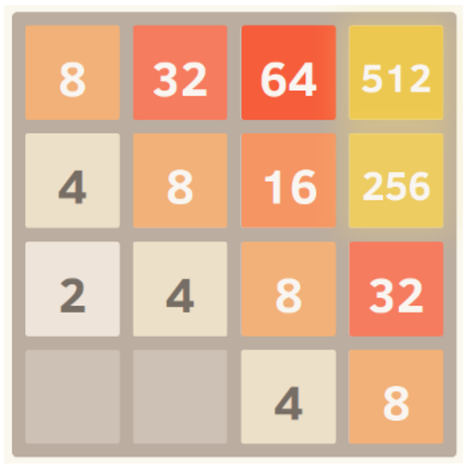
The edges heuristic increases the score for maintaining higher valued tiles on the edge of the board. This is a typical technique for human players since higher value pieces are difficult to combine making them more problematic in the center. The heuristic is implemented by adding the values of every edge to the score. Therefore the corners are weighted more and the heuristic prefers placing higher valued pieces on the corners.

5.3 MAXCORNER

This heuristic increases the score if the highest valued tile is in a corner, for the same rationale as the EDGES heuristic. The score increases by the value of the tile so this becomes more significant with higher valued tiles, especially since higher valued tiles are more difficult to merge.

5.4 MONOTONIC

Ensures that in each row or column the tile values are either increasing or decreasing by enforcing a ”penalty” for nonmonotic rows or comuns. . The goal of this heuristic to create a clustering of higher tile values in the corner of the grid as shown below:



A perfectly monotonic grid

This extends the previous heuristics by encouraging clustering higher valued pieces in a corner. This is implemented by looking at the difference between tile values in each row and column. If a row or column is not monotonic, the product of the difference between a tile minus the previous tile and the same tile minus the tile after it should be negative since the change in tile values are in opposite directions. The heuristic tracks the maximum absolute difference in each row and column. If a row or column is not monotonic, the difference valued is subtracted from the score. Therefore the monotonicity increases in significance with higher valued tiles.

```
if(MONOTONIC){
  int[][] prevDif=new int[4][3];
  // Check left/right directions
  for(int i=0;i<4;i++){
    for(int j=0;j<3;j++){
      prevDif[i][j]=game.tileAt(i,j+1)-game.tileAt(i,j);
    }
  }
  for(int i=0;i<4;i++){
    int maxDif=0;
    boolean nonMono=false;
    for(int j=0;j<3;j++){
      if(Math.abs(prevDif[i][j])>maxDif)
        maxDif=Math.abs(prevDif[i][j]);
      if(j>0)
        if(prevDif[i][j-1]*prevDif[i][j]<0)
          nonMono=true;
    }
    if(nonMono)
      score-=maxDif;
  }
  // Check up/down direction
  for(int i=0;i<4;i++){
    for(int j=0;j<3;j++){
      prevDif[i][j]=game.tileAt(j+1,i)-game.tileAt(j,i);
    }
  }...
```

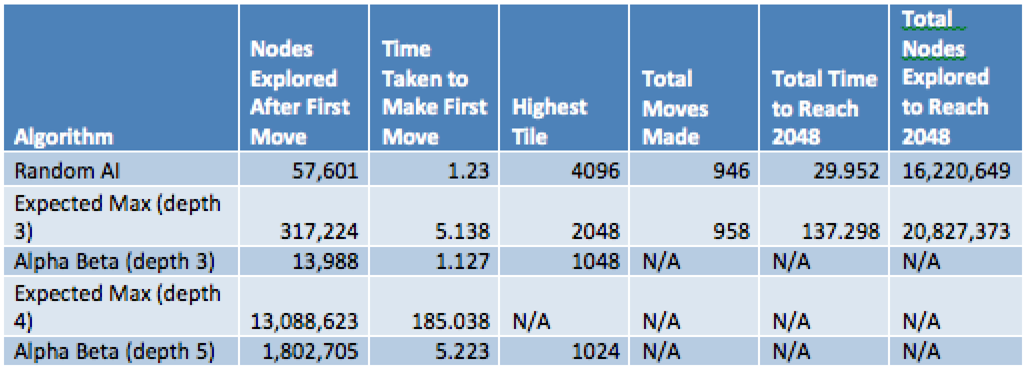
Implementation of the monotonic heuristic. The rest of the code for columns is the same as for rows and therefore omitted.

6 Comparing AIs

The different AIs varied in their performance. The Random AI was by far the fastest and explored the least number of nodes, but did not always reach 2048, but it was able to reach 2048 more than half the time. The randomness and basing utility on score make the AI appear to take unintelligent moves.

The ExpectedMax AI took significantly longer than the Random AI and increased exponentially in time and nodes explored with increased depth, due to high branching. This also posed a memory issue;the program ran out of heap memory at depth 4 after 50 or so moves. The use of heuristics significantly improved performance, but did not reduce the time cost of the algorithm. At depth 3 the expected max algorithm was able to reach 2048 a few times, but not consistently or much higher.

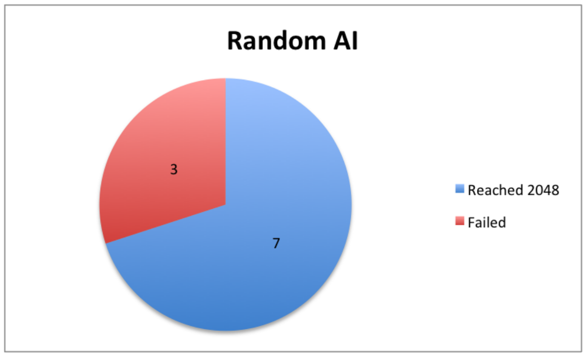
The AlphaBeta AI by far performed the worse of the three. It faced a similar branching problem as ExpectedMax AI, but to a lesser extent. Additionally the NOFOURS feature reduced run time and improved performance by testing tiles that more likely to occur. Even with a depth of 5, the Alpha Beta player was unable to achieve 2048 despite exploring over 17 million states.

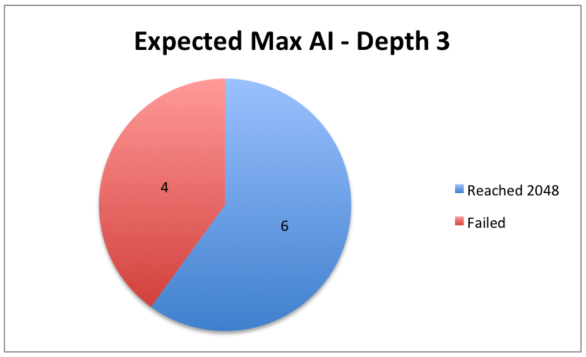


Algorithm	Nodes Explored After First Move	Time Taken to Make First Move	Highest Tile	Total Moves Made	Total Time to Reach 2048	Total Nodes Explored to Reach 2048
Random AI	57,601	1.23	4096	946	29.952	16,220,649
Expected Max (depth 3)	317,224	5.138	2048	958	137.298	20,827,373
Alpha Beta (depth 3)	13,988	1.127	1048	N/A	N/A	N/A
Expected Max (depth 4)	13,088,623	185.038	N/A	N/A	N/A	N/A
Alpha Beta (depth 5)	1,802,705	5.223	1024	N/A	N/A	N/A

Average data summary of different algorithms used.

In order to compare consistency of achieving 2048, I performed 10 trials for the random AI and the expected max AI and compared the percentage of times each algorithm reached the 2048 tile. The Random AI performed better than the Expected Max AI. This may be due to the greater depth at which the Random AI is able to search.





7 2048 Game Model

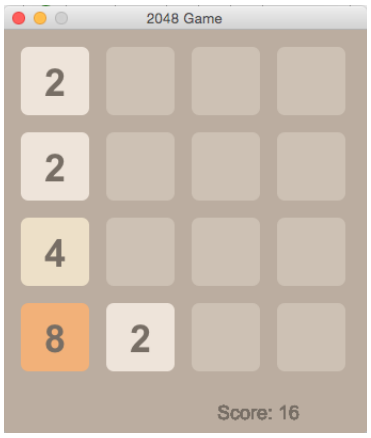
The project uses bulenkov's Java implementation of 2048, available on github, as the basis for the program with significant modifications. The program is organized into a model of the 2480 game in `Game2048.java`, a GUI View in `Game2048View.java` and a driver in `Game2048Driver.java`. A Player uses information about the state of the game to make decisions. The tiles are stored as an array of int where the index of the integer is the coordinates of the tile (from xy to x+y*4) and the integers themselves are the values of the tile.

Making a move in 2048 involves two primary parts: (1) moving all pieces in a particular direction so as to eliminate empty spaces and (2) merging adjacent pieces of the same value. For the left move, the moving is accomplished in `moveLine(int[] oldLine)`. The method parses through a line and of the game and adds non zero values to the queue then creates a new line starting with the non zero values in the queue and zeros for the other spot. The merging occurs in `mergeLine(int[] oldLine)`. The method similarly uses a queue, but also checks if the next value in the line is the same as the current value as it parses. It then adds double that value to the queue and skips the next value in the line (the duplicate value) in addition for checking for non zero values. For all other directions, the model rotates the board, performs the left move, then rotates the board back.

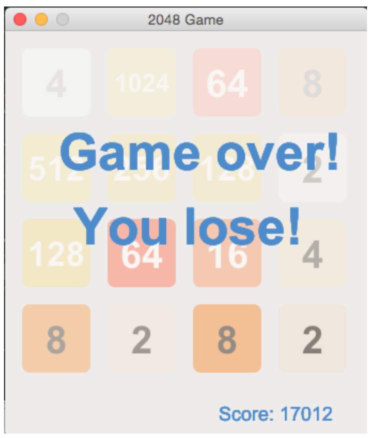
I also provided an `undoMove()` and `removeTile()` method to reduce memory use. I implemented these by using a stack of previous states (`prevMoves`) and previous scores (`prevScores`) represented by a `LinkedList`.

8 2048 Game View

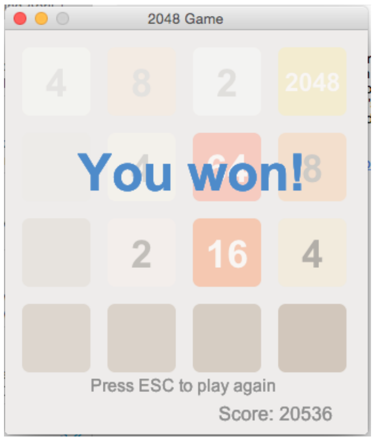
The game is displayed with GUI graphics in `Game2048View.java`. The project uses the java.swing package for graphics. The `drawTile()` method redraws tiles to reflect their updated location and updates the score. Since the `Game2048View(Game2048 game)` takes in the game in the constructor, a small delay is used before making the next move as the game's state changes to reflect the algorithm.



Score: 16



Game over!
You lose!
Score: 17012



You won!
Press ESC to play again
Score: 20536

Once the game ended in either a loss or a win, the view changes and informs the user whether a loss or a win was achieved.