Minimax for Playing the 2048 Game

Johannes Omberg Lier   Per-Christian Berg

November 24, 2015

1 General structure

We have implemented each state of the game as a class, *Node*, which most importantly consists of a board, a list of successor states and its probability. This probability is set when it is a chance node. Otherwise, it is None. The board is represented by a 4x4 matrix with the values of each cell representing the 'tiles'.

Functions worth mentioning related to nodes are *get_available_tiles*, which returns a list of positions on the board that is not occupied by a number. *assign_random_tile*, which adds the number two or four with the probabilities 0.9 and 0.1, respectively. *move*, which performs a move determined by a direction and a node. *find_obstacle*, which, for each tile, returns the farthest position available given a direction and the position one step further if it exists. This function is used to move and/or merge the given tile with another.

The main module of the program contains the *Ai-functions*. This module contains functions for traversing the expectimax tree, generating successors, calculating the heuristics and managing the game loop.

2 Expectimax

2.1 Expectimax tree

As mentioned earlier, each state has a list of successors. This way, we do not need to implement a tree structure. We simply generate all successors of a node and then recursively generate their successors until we reach the desired depth. When we reach a leaf node, its score is calculated and added to the score of the parent node. The score of a parent node is calculated differently depending on whether the node is a max node or a chance node. If the parent node is a max node, its score is the maximum score of all its successor. If the node is a chance node, the score is calculated as the sum of the successor's score times its probability. This implementation is a modified version of the *expectiminimax* algorithm, which have an additional opponent element added to the gameplay.

2.2 Prediction

The game is solved by continuously calling a prediction-function (called *find_best_child_node*). This function generates all possible successor nodes and can determine what move to perform by calling the expectimax function on all of the potential successors. The successor with the highest expectimax score will be the next node.

2.3 Pruning / Optimization

Since we implemented the expectimax algorithm, it is, as stated in the assignment, not an easy task to prune the tree. But since the probability of spawning a 4-tile is quite small, the probability that the best chance node is generated by spawning a 4-tile and that this 4-tile actually will spawn is very small. From this observation, when generating chance nodes, we generate one node for every available tile on the board and assumes the tiles all will have the value 2.

The depth of the search greatly impacts the run time. Because of this, we dynamically change the depth depending on the number of available tiles on the board. The more available tiles, the more successors needs to be generated and later inspected. And the number of total successors will increase exponentially for every level in the tree. The depth is defined as follows:

```
if   num_available > 7: depth = MAX_DEPTH - 2
elif num_available > 4: depth = MAX_DEPTH - 1
else:                   depth = MAX_DEPTH
```

Since each successor is a copy of the current root node with some additional changes, we made a huge improvement performance wise by working around the costly *deepcopy* function. We did this by making the node class take in a board as an argument and then only copy the root node's board into a new node instance. We can copy the board easily by just looping through it because we do not have a tile class that we initially planned, only integers.

```
temp_child = Node([row[:] for row in node.tiles])
```

2.4 Generation of successor states

We generate the successors in two different functions depending on what type of node the parent node is. We have divided this into the generation of max nodes and chance nodes as described in the exercise text. The first function, *generate_max_nodes*, generates boards by performing a move in all possible directions. After this comes the generation of chance nodes, where one successor for each available tile is generated. The spawn tile can spawn in every available place and it can be given the value 2 or 4. The function named *generate_chance_nodes* generates all different nodes where the tile can be spawned. Since we assume that the spawned tile has the same value the probability for a given chance node is:

$$prob = \frac{1}{NumberOfAvailableTiles}$$

This probability is used in the calculation of a node's score.

3 Heuristic

We have tried several heuristics and combinations between them, but in the final version of our program we only use one. We made a filter (a matrix), which forces the tiles on the board to form a "snake"-formation. This formation is achieved by having the biggest value in the top left corner of the filter and then alternating decreasing and increasing values for each row, starting with decreasing on the top row. The highest value in a row is less than the lowest in the row above. The heuristic score of a node is calculated by finding the dot product of the board and the filter. After some researching we found that the tiles merges in a fashion similar to a chain reaction when the tiles are in this formation. So this filter serves two purposes: Keeping the tiles with higher values in the top row and the tile with the highest value in the corner so they are not in the way of tiles with lower values which needs to merge several times.

$$filter = \begin{pmatrix} 10 & 8 & 7 & 6.5 \\ 0.5 & 0.7 & 1 & 3 \\ -0.5 & -1.5 & -1.8 & -2 \\ -3.8 & -3.7 & -3.5 & -3 \end{pmatrix}$$

$$good\_board = \begin{pmatrix} 512 & 128 & 0 & 0 \\ 128 & 16 & 32 & 0 \\ 32 & 64 & 0 & 0 \\ 16 & 8 & 0 & 0 \end{pmatrix}$$

$$bad\_board = \begin{pmatrix} 32 & 128 & 512 & 128 \\ 16 & 0 & 16 & 32 \\ 0 & 0 & 64 & 0 \\ 0 & 0 & 8 & 0 \end{pmatrix}$$

$$heuristic\_value(good\_board) = filter \cdot good\_board = 6048.8$$

$$heuristic\_value(bad\_board) = filter \cdot bad\_board = 5736.8$$

We can see here that good_board is considered the better choice because the value of its tiles are placed in a formation more similar to the filter. Notice that these two boards are successors from the same node.

We considered adding points to the score based on the number of available tiles on the board, but did not find a good way of adding it to the existing score. It should contribute positively if done correctly, but since we achieve the 2048 tile frequently without it we did not prioritize it. Shown below is an example of how we could implement it.

$$heuristic\_value(board) = filter \cdot good\_board + num\_available\_tiles^2$$