2048 Expectimax Solver

Anders Sildnes, Andrej Leitner *students of NTNU*

***Abstract*—This text answers assignment 5: writing a solver for the game 2048. The purpose of the game is to slide tiles with numbers 2^n, and join them together to form 2048 (2^{11}), or possibly higher. We present a solver using Expectimax algorithm. We explain our choice of heuristics and results.**

2048 can be thought of as a 2-player, turn-based game. The opponent places tiles valued either 2 or 4 in an availible location. Then, the other player makes a move to slide all tiles in a given direction. This goes back and forth until there are no more availible moves.

In a turn-based game, you have the time to consider the consequences for each possible action. This gives computers immense advantages over humans: IBM's chess-solving machine "Deep Blue" won against Garry Kasparov in 1997 considered more than 200 million possible moves per second[1].

In the case of 2048, this could yield a valid solution in a short amount of time. However, not everyone has access to such fast hardware and multi-threading so a way to prune the search space is needed.

Minimax Algorithm

The purpose of minimax is to minimize loss for a worst possible scenario. This means choosing an action such that your opponent can do "least amount of damages", i.e. leaving you in a promising state. This is also called *adversial search*. For 2048 this would mean always making a move such that no matter where the next tile goes, and no matter what value, you will still find a way for success.

[1]For more, see http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

2048, however, is a stochastic game. The opponent will never purposly select a bad tile, nor *choose* a bad value. This is to say that there is no opponent, so it is possible to always land a best case scenario. Therefore, we felt that minimax, considering the worst case, is too pessimistic in its search space. Furthermore, the distribution of numbered tiles are given: there is a 90% chance of getting a tile valued 2, and 10% chance of getting a tile valued 4. The location is random. Also note that in some cases, a tile in position i, j is no different from having a tile in position $i+i, j$ if the next move it to slide the tiles downward. Thus the possible search space is not too big and one can use statistics to get accurate enough resuls.

Using stochastic information in adversial search leads to expectimax. After expanding the search space up to a given depth, you can back up from each node and introduce a chance node. The chance nodes multiply the value to each of its children (given a state, the objective value to each of the children). The chance nodes are used when pruning the search space. Their importance is that unlikely values will yield a lower penalty to the overall score of the search node, such that the node is not necessarily pruned away.

Heuristic

To assess whether or not a current game state is good or not, we have different objective functions to assess game state. Using these to advance in the solution space is called heuristical search. Our heuristics are developed based on ideas we found online and through some play of our own.

Gradients

If a large value is in the middle of the board, it will be hard to pair up with other adjacent tiles. Therefore, we would like to set a natural preference for having larger values clustered in a corner. This way, they do not interfere with other lower-valued tiles. Next to the highest value tile, you want other, lower valued cells. This extends throughout the board. However, having large values in two corners implies that we have two (or more) tiles with large values, separated by lower tiles. Therefore, we penalize boards where large values are far apart.

To calculate the value of a board, we apply a 4x4 mask and retrieve the sum of adding all the values in the masked board. The mask is shown Figure 1. Each number in the tiles represents a scalar which is multiplied with the value from the board. Since large value might be clustered in all corners, we apply 4 masks, one for each corner respectively. Then, the largest resulting value is chosen as the one to be used as a value for the state.

Why not use more heuristics?

Many other heuristic functions exist:

1) **Board score** – whenever two cells merge, the board score increases.
2) **Free tiles** – giving scores based on how many cells are left.
3) **Non-monotnic rows/coumns** – prefer boards where there are large, but similar tiles.
4) *Possible merges* – prefer boards where many merges are possible, i.e. boards with tiles of the same value.

3	2	1	0
2	1	0	-1
1	0	-1	-2
0	-1	-2	-3

Fig. 1: Our gradient mask

Heuristic 1 is difficult to implement in practise. When greedily preffering boards with higher tiles you might exclude boards with high merges. Heuristic 2 can also be used. Arguably, it is already used in our gradient function: boards with tiles in opposite corners are penalized. Therefore we did not see the need to implement this heuristic expliclitly. Furthermore, there are cases where having many free tiles not necessarily is a bad thing. The two last heuristics are shown to

Exepctimax has execution time $O(b^m)$. Here, b is the *branching factor*, i.e. the number of possible children from a given board. For each move, $0 \leq b \leq 4$. m is the depth of the search tree. It can easily be seen that this execution time quickly grows: for a search depth of 6, the execution time is 4096 for each possible move. In addition, we have to apply our mask in Figure 1 four times and introduce chance nodes. Therefore, to avoid cluttered code and execution, we chose to only rely on the gradient method. However, if this assignment had rewarded execution time and scores higher than 4096, we would implement more of the heuristics mentioned above.