2048 Expectimax Solver

Anders Sildnes, Andrej Leitner *students at NTNU*

***Abstract*—This text answers assignment 5: writing a solver for the game 2048. We present a solver using Expectimax algorithm. We explain our choice of heuristics and results.**

THE purpose of "2048" is to slide tiles with numbers 2^n, and join them together to form 2048 (2^{11}), or possibly higher. Between each slide, i.e. moving all the tiles on the board north, west, east or south, a random tile valued either 2 or 4 appears on the board, which adds difficulty in solving the game.

Assumptions

It is possible to obtain tiles values up to 2^{14} within reasonable time. Our assignment bounds, however, did not require us to achieve such a high level of performance:

1) The highest value tile does not need to exceed 4096 (2^{12}).
2) Execution time is not critical (acceptable as long as < 10 minutes)

This implies that our chosen heuristic and solver method does not need a strong level of optimization to work. Indeed, the work of (1) shows that a naïve search can be used to solve the game within the bounds listed above. Still, we have implemented an expectimax approach which is much more scalable.

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 damage", i.e. leaving you in a promising state. The process of determing the best action availbile given your opponent's possible re-action is also referred to as *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, your action minimizes loss.

2048, however, is a stochastic game. The opponent will never inteligently select a bad location or value. Therefore, we felt that minimax, which always prepares for the worst case, is too pessimistic in its choice of actions. Furthermore, the distribution of tiles are given: there is a 90% chance of getting a tile valued 2, and 10% chance of getting a tile valued 4. Thus, it is possible to make somewhat accurate predictions about the expected likelyhood of future states of the game. What cannot be predicted is the location of new tiles. However, the possible locations are typically not many enough so that an adversial search cannot completely explore the possible outcomes.

Minimax with probabilistic values to each state leads to the modified *expectimax* algorithm. Search space is expanded similar to in minimax, but it is extended with *chance nodes*. The chance nodes considers each possible successor states, but instead of considering the value from each board/objective function, it also takes into account the probability of getting each state. Their importance is that if an action might lead to a bad, but unlikely result, this result will not significantly reduce the overall value of the action.

Heuristic

To assess whether or not a current game state is good or not, we have one heuristic function that assesses how promising a given board is. Our heuristic is 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 4×4 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.

Formally, the gradient calculation can be calculated as follows:

$$
\begin{aligned}
&\textbf{Let}\\
\text{board}_{i,j} &= 4 \times 4 \text{ square lattice}\\
&\quad \text{s.t. board}_{0,0} \text{ represents north west}\\
dec(x) &= \{x, x-1, x-2, x-3\}\\
inc(x) &= \{x, x+1, x+2, x+3\}\\
g_{NW} &= dec(3), \ldots dec(0)\\
g_{NE} &= dec(0), \ldots dec(3)\\
g_{SW} &= inc(-3), \ldots inc(0)\\
g_{SE} &= inc(0), \ldots inc(-3)\\
G &= \{g_{NW}, g_{NE}, g_{SW}, g_{SE}\}\\
&\textbf{Then:}\\
&\textit{For a given board:}\\
\text{gradient } &= \underset{\forall g \in G}{\text{argmax}} \sum_{i,j} g_{i,j} \cdot \text{board}_{i,j}
\end{aligned}
$$

Furtherly, to each chosen gradient value, we calculate $E[\text{board}]$, i.e. the expected value, given the value of the board and the probability. This is given by the probability of having a tile valued either 2 or 4. We assume there is a uniform probability of tile location (so this probability is 1). The values are multiplied together, and when combined make up the heuristical assessment value for each board.

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

Fig. 1: Example gradient mask (g_{NW})

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.

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 $O(b^m n^m)$. Here, b is the *branching factor*, i.e. the number of possible children from a given board. For each state, $0 \leq b \leq 4$. m is the depth of the search tree. n is the number of possible moves, i.e. possible tiles with value either 2 or 4. It can easily be seen that this execution time quickly grows: for a search depth of 6, the execution time is $4^6 \cdot n^4$ 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.

Structure of source code

We implemented our solution in Java. This is because both members of the group were familiar with the language. The GUI is rendered through a JFrame, and the tiles are drawn using AWT's Graphics2D.

There is a builtin keylistener that reads for keyboard input. However, one can also enable autosolving, which toggles a flag in the keylistener and solves the board by itself. The builtin solver works as follows:

1) For each possible move (left,right,up,down) by the player, call "computers"
2) "computer": For each possible tile placement:

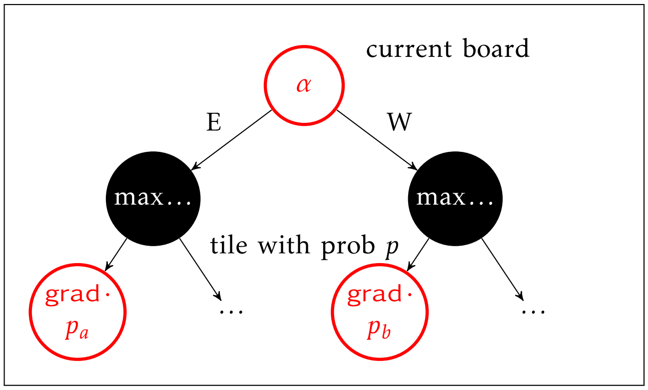

Fig. 2: Example expectimax tree, with implicit chance nodes. Each black cell is a board after a move is made (waiting for opponent to place a tile). The black cells' value is the highest value of its children.

a) place a tile
b) call all possible player moves, decrementing depth by 1

3) calculate E[current state]
4) … goto step 1 if depth > 0
5) Get E[current node], multiply upward to root
6) Once complete k-depth=tree is complete, choost most promising branch.

To realize the above code, a few classes are needed. We implemented each possible board as a class with each tile as a subclass. It could be noted that since each tile will not need to exceed $2^{16} = 65536$, a 64-bit architecture can represent one row as a 64-bit number (16×4) which permits fast bitwise operations for altering the board. However, without classes (objects) the GUI is more difficult to render, so we chose not to use this apporach, even though it allows fast board checks.

Results

Our highest result achieved so far is 8192. On average, we achieve 4096 in 80% of the cases, in < 5 minutes. Figure 3 shows this result and our board.

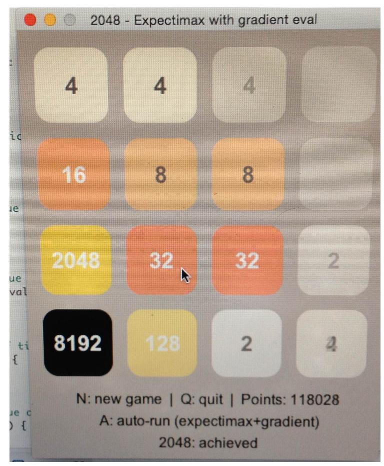

Fig. 3: Example solver and board

References

[1] "ronzil" *2048-AI*, *https://github.com/ronzil/2048-AI*. GitHub