IT3105 module 4

Expectimax for playing the 2048 game

Iver Jordal

Expectimax

The program can become slow if it searches deeply and/or has a large branching factor. There's a trade-off:

- Large search tree: Slow execution, but good end results
- Smaller search tree: Faster execution, but worse end results

I implemented the game and AI in Python, and I was not able to run the AI with a depth of 4 in a timely manner. Based on experimentation, a depth of 3 (where each level has a max node *and* an average node) and a branching factor of 4 yields a fast-running program, while the 2048 tile is achieved in around 50% of the runs, which is good enough for this assignment.

In order to keep the branching factor down, the number of possible spawning events in CHANCE nodes is limited to 4. In other words, up to 4 spawning positions are chosen. In fact, only twos are spawned. If fours are also considered, then the branching factor is doubled, and this significantly slows down the program. In MAX nodes, every possible move (direction) is considered. Essentially, my expectimax implementation is merely an approximation of the "proper" expectimax algorithm.

I have not attempted to prune nodes like in minimax with alpha-beta-pruning. However, when a "dead" node (i.e. with no possible moves) is found, that node returns a zero for its heuristic value, and the search will not (and can not) go any deeper from there. This way, the AI is less likely to choose moves that might lead to the end of the game.

The expectimax function is implemented as two recursive functions. MAX nodes will call the expectimax_average() function on each possible move. CHANCE node will call expectimax_max() on each spawning event, given that the max depth has not been reached. If the max depth has been reached, get_heuristic() will be called instead of expectimax_max(). This way, the heuristic value is calculated only for leaf nodes.

Heuristic function

In a game of 2048, there are a lot of things to consider. Therefore, my heuristic function calculates the sum of six terms that each give some positive weight to specific qualities in a state of the game.

$$\begin{aligned} &Heuristic\ value \\ &\quad = Gradient\ cell\ weight + \ Empty\ cells + \ Smoothness + \ Monotonicity \\ &\quad + \ Max\ tile\ value + \ Max\ tile\ in\ corner \end{aligned}$$

Next, I will go into detail about why each term is important and how each term is calculated.

Gradient cell weight

In 2048, it is a good idea to build against one of the sides, i.e. generally keep the largest tiles on one side rather than having them scattered around. The cell weight concept is to calculate a variable that is the sum of each tile value times a weight that is specific to that position. For example, here is the gradient matrix that gives more weight to tiles in the upper area:

$$w_{up} = \begin{matrix} 4 & 4 & 4 & 4 \\ 3 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 \end{matrix}$$

However, in bad states it is good for the AI to have some flexibility in choosing a new side to build against, rather than enforcing the same side as before. Therefore, I figured that it's a good idea to calculate one score for each gradient direction, and then choose the maximum of the four scores to be the cell weight term.

Further, it's obviously good to favor few large tiles over many medium tiles. Therefore, I've modified the concept slightly: each tile value is raised to the power of 1.3 before it's multiplied with the cell weight.

$$score_{up} = \sum c_{i,j}^{1.3} * w_{up}{}_{i,j}$$
$$score_{right} = \sum c_{i,j}^{1.3} * w_{right}{}_{i,j}$$
$$score_{down} = \sum c_{i,j}^{1.3} * w_{down}{}_{i,j}$$
$$score_{left} = \sum c_{i,j}^{1.3} * w_{left}{}_{i,j}$$

Where $c_{i,j}$ is defined as the value of the tile in the i-th row in the j-th column and where w_{right}, w_{down} and w_{left} are created by the same concept as w_{up}, i.e. a matrix with values ranging from 1 to 4 according to a gradient in the given direction.

Finally, $Gradient\ cell\ weight = \max(score_{up}, score_{right}, score_{down}, score_{left})$

Empty cells

I added this term to avoid a cramped situation with a small number of empty tiles. Such a situation is bad because it generally means fewer possible moves. No possible move means game over. Therefore, larger scores are given to states with a larger number of empty tiles.

$$Empty\ cells = 0.05 * c_{max} * (\#empty)^2$$

Where c_{max} is defined as the value of the largest tile and $\#empty$ is the number of empty tiles on the board. c_{max} is a factor because all terms in the heuristic value are scaled with the tile values.

Smoothness

Two adjacent equal tiles means that they can be merged, and that is a good thing. The concept of smoothness tries to boil this down to a number. This number becomes larger when there are more equal tiles adjacent to each other. Smoothness is calculated for each row and for each column, and finally those numbers are summed to become the smoothness term.

$$smoothness_{row\ or\ column}(r) = \sum_{k=1}^{3} f_s(r_k, r_{k-1})$$

Where $f_s(a,b) = \begin{cases} \min(b,4)\ if\ a = b \\ 0\ otherwise \end{cases}$ and r_k is the value of the k-th tile in the row or the column.

$$Smoothness = 0.5 * \left(\sum_{each\ row\ as\ r} smoothness_{row\ or\ column}(r) + \sum_{each\ column\ as\ r} smoothness_{row\ or\ column}(r) \right)$$

Monotonicity

Monotonicity is the concept of having a row or a column where the values are ordered in either ascending order or descending order. The idea is to put tiles that are almost the same next to each other, because when a tile gets merged into the smallest of the two, then the two tiles become equal, and are ready to merge. This way, having a monotonic row or a column means that one can merge many tiles consecutively in order to get a single, large tile. For example, the row [256, 128, 64, 64] can get a 512 tile by moving left three times.

$$Monotonicity = \sum_{each\ row\ as\ r} monotonicity_{row\ or\ column}(r) + \sum_{each\ column\ as\ r} monotonicity_{row\ or\ column}(r)$$

$$monotonicity_{row\ or\ column} = \left| \sum_{k=1}^{3} f_m(r_k, r_{k-1}) \right|$$

$$f_m(a,b) = \begin{cases} 0\ if\ a = b \\ f_{log2diffscore}(a,b)\ if\ a > b \\ -f_{log2diffscore}(a,b)\ if\ a < b \end{cases}$$

$$f_{log2diffscore}(a,b) = \frac{a+b}{abs\left(f_{log_2}(a) - f_{log_2}(b)\right) + 1}$$

Observe that in the calculation of monotonicity, a smaller score is given for ordered values that have a large gap in difference, due to the way $f_{log2diffscore}$ works.

$$f_{log_2}(x) = \begin{cases} 0\ if\ x = 0 \\ log_2(x)\ otherwise \end{cases}$$

Max tile value

This term is simply the value of the largest tile in the current state. It's useful because it gives extra weight to moves that merge two tiles to get a tile that is larger than all the others.

$$\text{Max tile value} = c_{max}$$

Max tile in corner

It's generally a good idea to have the largest tile in a corner, because that way it becomes easier to get monotonic rows/columns. This term gives such states extra weight.

$$\text{Max tile in corner} = \begin{cases} 0.2 * c_{max}, \text{ if a tile with value } c_{max} \text{ can be found in a corner} \\ 0\text{ , otherwise} \end{cases}$$