IT3105 Module 4

Johan Slettevold
Iver Egge

October 2015

1 Expectimax

1.1 Pseudocode for expectimax

The expectimax algorithm is implemented as a recursive function following the pseudocode below:

```
expectimax(node, depth, isMaximizingPlayer):
    if depth = 0 return heuristic value of node

    if isMaximizingPlayer:
        bestValue = negative infinite
        for child of node:
            value = expectimax(node, depth - 1, false)
            if value > bestValue:
                direction = direction of child
        return direction

    if not isMaximizingPlayer:
        totalValue = 0
        for child of node:
            totalValue = expectimax(node, depth - 1, true)
        totalValue = totalValue / number of children of node
        return direction of node
```

1.2 Children

When a node is asked to generate children for the **maximizing player**, it produces 1-4 new nodes based on the legal directions that the node can move.

However, when a node is asked to generate children by **chance**, it generates a node for each possible tile spawn. We assume that random tiles will have the value 2 and ignore the possibility of a 4, resulting in C rather than 2*C chance

nodes. The score of a chance node is the average heuristic score of all its children.

Pseudocode for generating the chance nodes:

```
generateChildren:
(...)
if not isMaximizingPlayer:
    for i = 0 to number of empty tiles on board:
        add new node with grid from generateChanceNodes(i) to children
    return children

generateChanceNodes(fillTile):
    tileCount = 0
    expectGrid = copy(grid)
    from i = 1 to 4:
        from j = 1 to 4:
            if grid[i][j] = 0:
                if tileCount = fillTile:
                    expectGrid[i][j] = 2
                    return expectGrid
                tileCount + 1
    return
```

1.3 Dynamic depth

We set the depth of the search tree to 7 layers as long as there are more than 5 empty tiles. When there is less than 5 empty tiles we adjust the depth to 9 layers. We did this to allow the algorithm to be more careful when approaching a losing situation. The depth values 7 and 9 layers are based on what seemed suitable for running the algorithm in reasonable time on off-the-shelf computers.

2 Heuristic

To calculate the heuristic value of a leaf node we run through all of the tiles of the node and add to the total heuristic the value of the tile times a pre-defined score based on the location of the tile. The goal of the heuristic function is to value a node with the tiles placed incrementally in a "snake" pattern very highly. A snake pattern was chosen because it optimally lines up equally sized tiles next to each other, with the largest value placed in a corner.

score(N) is the heuristic function based on a grid N.
$\mathbf{T_{i,j}}$ is the value of the tile with placement (i,j) in grid N.
$\mathbf{W_{i,j}}$ is the weight of the value with placement (i,j) in N.

$$score(N) = \sum_{i=1}^{4} \sum_{i=1}^{4} T_{i,j} \cdot W_{i,j} \tag{1}$$

The weight matrix W is defined as:

$$W = \begin{bmatrix} 32768 & 16385 & 8192 & 4096 \\ 256 & 512 & 1024 & 2048 \\ 128 & 64 & 32 & 16 \\ 1 & 2 & 4 & 8 \end{bmatrix} \tag{2}$$

The values in the weight matrix W is computed from 2^n starting in the lower left corner and continuing upwards in a "snake"-like pattern (first right then left then right etc.).

2.1 Approaches

When running the algorithm, we noticed a dramatic effect of small values getting isolated in the upper right corner (W(4,1)). To prevent this from happening, we tried to reweight the grid in these situations. The most interesting experiment was to lower the weight of the tile under the isolated tile (W(4,2)) and at the same time heightening the weight of the tile directly to the left of that tile (W(3,2)), whenever such an isolation occurred. One could view this as a slight "gradient"-style pattern modification. After a series of tests, this heuristic seemed less effective than the standalone "snake" pattern, and thus we discarded it.

In addition to trying to solve this rather specific problem, we experimented with more general heuristics; Such as looking at the number of open tiles, the number of mergeable tiles, and the value of the difference in tile size of the rows or columns of the grid. These heuristics were manually applied and tested with weights which gave them both larger and lesser impact than the "snake" pattern. None of these noticeably improved the results, and therefore, none of them were applied to the final solution.

3 Result

Our algorithm successfully solves the game (acquiring the 2048 tile) in the majority of its runs. The implementation does produce 4096 tiles quite often, and has been recorded to reach the 8192 tile several times. In conclusion we are quite happy with the implementation and results.