Ezdin Aslancı - 150112022
Kutalmış Coşkun - 150113018

1. About Project

   In this project, we have implemented an AI player for the well-known 2048 single-player sliding block puzzle game designed by Italian web developer Gabriele Cirulli.

   We have used expectiMax algorithm with 4 different heuristic functions.

   We have used 64-bit C++ with Armadillo linear algebra library to implement both the game and AI.

   Since we did not use a GUI, we present our results taken from console.

2. Heuristic Functions

   i. Number of Empty Cells

   The game continues until there is no empty cell left on the grid and there is no possible action to take. This makes maximizing the number of empty cells is a nice strategy for both avoiding stuck positions and merging tiles that have the same value.

   However, this approach makes the AI to rush for merging tiles and it sometimes fails to form tiles with higher values since the utility taken from a 2+2 merge is the same as taken from a 512+512 merge.

   `numOfEmptyCells: The number of zeros on the grid.`

   Here are the best results we obtained using this heuristic:

1 PLY			
2	4	8	2
4	8	32	4
8	32	128	8
16	128	256	512

2 PLY			
2	4	8	4
512	32	1024	16
16	256	32	4
8	128	16	2

3 PLY			
4	128	8	4
16	1024	256	128
4	512	32	16
2	4	2	8

   ii. Monotonicity of Rows and Columns

   It is nice strategy to cluster close valued tiles together since the possibility to merge those is higher if they are located near on the grid. So we have implemented a heuristic function that returns higher utility values if the rows and columns are ordered monotonically with smaller log distances between consecutive tiles.

   This makes `[2048, 1024, 512, 256]` a more valuable row than `[2048, 8, 4, 2]` although they are both monotonic.

   We have also taken into account the fact that the tiles gathered on the edges of the grid are more valuable. So we doubled the utility taken from being monotonic if the row or column is located on one of 4 edges.

   `orderScore =`

$$\sum_{allrows}\sum_{all\,cols}\begin{Bmatrix} monotonicity, & \text{if vector is monotonic} \\ 0, & \text{if all elements are equal} \\ -1, & \text{if vector is not monotonic} \end{Bmatrix}\times\begin{Bmatrix} 2, & \text{if vector is located on edge} \\ 1, & \text{otherwise} \end{Bmatrix}$$

   `where,`

$$\texttt{monotonicity} = \begin{cases} \dfrac{\sum elements}{\sum pair-wise \log distances}, & if\ vector\ is\ monotonic \\ 0, & if\ all\ elements\ of\ vector\ are\ equal \\ -1, & if\ vector\ is\ not\ monotonic \end{cases}$$

Here are the best results we obtained using this heuristic:

1 PLY			
32	64	128	512
8	16	32	128
4	8	16	32
2	4	8	2

2 PLY			
4	2048	8	2
32	256	32	64
4	16	128	16
2	4	32	2

3 PLY			
2	32	2	4
4	16	64	32
8	1024	32	4
4096	8	4	2

iii. Snake Formation

The best strategy for placing tiles on the grid is making a snake like formation. With this formation, merges can propagate through the snake and it becomes more likely to form higher valued tiles at the head of the snake (the head is located on one of the corners).

We achieved this by generating snake matrices and picking the best snake according to the sum of elements on the matrix that is obtained by pair-wise multiplying the grid and the snake matrix.

Here are some snake matrices:

$$\begin{vmatrix} 13 & 14 & 15 & 16 \\ 12 & 11 & 10 & 9 \\ 5 & 6 & 7 & 8 \\ 4 & 3 & 2 & 1 \end{vmatrix} \quad \begin{vmatrix} 1 & 2 & 3 & 4 \\ 8 & 7 & 6 & 5 \\ 9 & 10 & 11 & 12 \\ 16 & 15 & 14 & 13 \end{vmatrix} \quad \begin{vmatrix} 16 & 9 & 8 & 1 \\ 15 & 10 & 7 & 2 \\ 14 & 11 & 6 & 3 \\ 13 & 12 & 5 & 4 \end{vmatrix} \quad \begin{vmatrix} 4 & 5 & 12 & 13 \\ 3 & 6 & 11 & 14 \\ 2 & 7 & 10 & 15 \\ 1 & 8 & 9 & 16 \end{vmatrix}$$

There are 8 different snake matrices. (4 corners * 2 directions)

In order to make the difference between the head and the tail of the snake, we also used exponentials, for example:

$$\begin{vmatrix} 2^{13} & 2^{14} & 2^{15} & 2^{16} \\ 2^{12} & 2^{11} & 2^{10} & 2^{9} \\ 2^{5} & 2^{6} & 2^{7} & 2^{8} \\ 2^{4} & 2^{3} & 2^{2} & 2^{1} \end{vmatrix} \quad \begin{vmatrix} 2^{1} & 2^{2} & 2^{3} & 2^{4} \\ 2^{8} & 2^{7} & 2^{6} & 2^{5} \\ 2^{9} & 2^{10} & 2^{11} & 2^{12} \\ 2^{16} & 2^{15} & 2^{14} & 2^{13} \end{vmatrix}$$

a. Constant Snake Formation with Pairwise Distance Minimization and Order Penalty

In this heuristic, the main difference with 4th heuristic is that the AI only uses snakes which occur only bottom-right positions. Which are:

$$\begin{vmatrix} 4^3 & 3^3 & 2^3 & 1^3 \\ 5^3 & 6^3 & 7^3 & 8^3 \\ 12^3 & 11^3 & 10^3 & 9^3 \\ 13^3 & 14^3 & 15^3 & 16^3 \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} 4^3 & 5^3 & 12^3 & 13^3 \\ 3^3 & 6^3 & 11^3 & 14^3 \\ 2^3 & 7^3 & 10^3 & 15^3 \\ 1^3 & 8^3 & 9^3 & 16^3 \end{vmatrix}$$

Here is the overall formula for this heuristic:

$$\texttt{utility} = \texttt{snakeScore} - \texttt{sumOfPairwiseDistances}^3 - \texttt{penalty}^2$$

where,

snakeScore: Reward value that takes maximum value when grid becomes one of the snakes above.

sumOfPairwiseDistances: Sum of pairwise distances between all neighbors on the grid.

penalty: If maximum 4 values are not sorted at the corner, it gives a penalty value for each unsorted tile.

Here are the best results we obtained using this heuristic:

1 PLY			
8	2	4	2
2	8	16	8
16	64	32	16
2	16	128	1024

2 PLY			
2	8	2	8
32	16	32	2
4096	64	128	64
2	2048	512	256

3 PLY			
4	2	4	2
2	8	32	4
32	64	128	64
8192	4096	1024	512

b. Dynamic Snake Formation with Monotonicity and Empty Cell Maximization

In order to achieve better results, we combined few heuristic functions with different scalar coefficients.

AI finds the one among all possible snake matrices and can dynamically switch from one formation to another. We also used monotonicity and empty cell maximization approaches in this heuristic function.

Here is the formula we used:

```
utility = snakeScore + orderScore + (30 * numOfEmptyCells)
```

where,

snakeScore: Score taken from pair-wise multiplication of the grid (game state) and the best snake matrix.

orderScore: Score taken from monotonicity.

numOfEmptyCells: The number of zeros on the grid.

Here are the best results we obtained using this heuristic:

1 PLY			
512	2	8	2
256	8	64	4
32	128	16	8
8	32	4	2

2 PLY			
4	2048	512	256
4096	128	256	16
64	32	2	4
4	2	4	16

3 PLY			
8192	2	16	2
4	64	32	8
128	256	4	512
2	2048	1024	4

3. Results

Here are the results we gathered by playing the game 20 times for each configuration:

Max of Max Tiles Found in All Experiments				
	Heuristic 1	Heuristic 2	Heuristic 3	Heuristic 4
1-ply	512	512	1024	512
2-ply	1024	2048	4096	4096
3-ply	1024	4096	8192	8192
4-ply				

Min of Max Tiles Found in All Experiments				
	Heuristic 1	Heuristic 2	Heuristic 3	Heuristic 4
1-ply	128	256	128	128
2-ply	128	512	1024	512
3-ply	512	1024	2048	2048
4-ply				

Average of Max Tiles Found in All Experiments				
	Heuristic 1	Heuristic 2	Heuristic 3	Heuristic 4
1-ply	268.8	294.4	345.6	313.6
2-ply	633.6	921.6	1996.8	2278.4
3-ply	896	1484.8	4608	4300.8
4-ply				