Solving 2048 with Expectimax

David Cook

December 2, 2022

Contents

Abstract

The simplest goal of this project is to play the puzzle game 2048 with the expectimax algorithm. I intend to make it able to play any size of the 2048 game, though eventually, any algorithm will become impractical if the grid is too large. To mitigate this, I intend to find ways of pruning the tree as much as possible. Officially the game of 2048 ends when the tile of 2048 is achieved, the game can be continued until no more moves are possible. A video of the program running can be found here [1].

1 Introduction

1.1 The Problem

2048 is a game which features a 4x4 grid containing powers of 2. Each game starts with two cells, with a value of either 2 or 4. All the tiles can be slid in any of the four directions (up, down, left and right); all the tiles can move simultaneously. When two of the same powers of 2 collide, the tiles merge, creating the next power of 2. The new tile increases the score by its new value. After each move, a new tile (either 2 or 4) will appear in a random free cell [2].

The goal of this project is to play a 2048 game using an AI-search algorithm. Some previous projects have reached the tile of 32k[3]. Reaching 32k is uncommon, many expetimax implementations are only capable of reaching 4096 [4][5].

A promising search algorithm is expectimax [4]. This is an algorithm designed for scenarios where there are two agents, one making random decisions and one making rational decisions [6, p.200]. In some situations, it is possible to pre-compute a decision tree, saving execution time. It is not practical to pre-compute a traditional 2048 decision tree, particularly when factoring in variable-size games.

Expetimax is also the algorithm used by [3], the implementation capable of reaching a tile of 32k, however this is optimised to the point were the code is difficult to understand, for example the game's state appears to be stored an a 64-bit unsigned integer, and modified using bit wise operations.

1.2 Deliverables

1.2.1 Proof of concept programs

1. Decision Tree
2. Simple Expectimax Example
3. 2x2 2048 Game
4. 2048 with a heuristic

1.2.2 Report

1. Professional Issues: Licensing. (interim report)
2. Design patterns in AI search. (interim report)
3. Techniques used by humans and previous automated solvers. (interim report)
4. User interface design for the solver. (interim report)
5. Complexity and big O notation. (interim report)
6. NP Hardness
7. The practicality and the effectiveness of the pruning expectimax tree.
8. The heuristics have been generalised to support more 2048 games.
9. Describe interesting algorithms and programming techniques, such as expectimax, used in the project.
10. Implementation and performance of the decision tree.

The report will describe the implementation and optimisations of the expectimax algorithm, focusing on any software engineering principles used in the processes.

1.2.3 Final Program

The final program will:

1. Be written in java, with a full Object-oriented design, using modern software engineering principles.
2. Be theoretically capable of playing any $n \times m$ 2048 game, though eventually break down due to performance.
3. Have a user interface capable of keeping track of stats about the algorithms and an easy way of creating new $n \times m$ games.

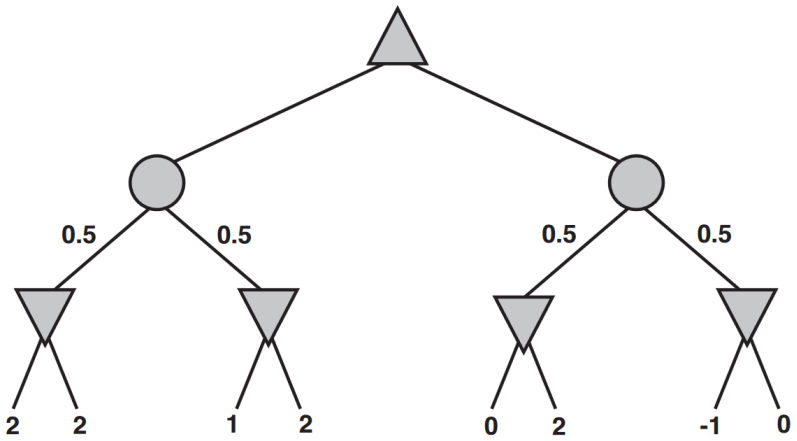

Figure 1: An expectiminimax tree example [6, p. 200]

2 Proof of Concepts

2.1 Decision Tree

There are many ways of implementing a tree structure; various ones are more appropriate than others. The operations that are required in this situation are [6]:

- Generate a tree from the root node.
- Transverse the entire tree to calculate the score.
- Walk to a direct child node and regenerate the tree.

In the later case, the scores will need to be regenerated on the pre-existing nodes and some new nodes node to generate.

I will assume generating a node can be done in $O(1)$ time; however, this may not be true. Generating or traversing nodes in a tree with n nodes can not be done in $O(n)$ time. Reading a direct child can not be done in $O(1)$ time.

A common approach to making a tree is using a simple node class such as:

```
public class Node {
    float score;
    Node[] children;
}
```

This is a node with an array of references to its child nodes. Traversing this entire tree will have a time complexity of $O(n)$. Arranging these nodes into a tree structure takes $O(n)$ and finally picking a direct child of a node takes $O(1)$, This simple implementation of a tree is adequate for the job, so more complex tree solutions are not necessary.

2.2 Expectimax

The expectiminimax algorithm allows for games to play with two rational agents and an element of chance [6, p. 200]. A simplified version of this algorithm, using only one rational agent, called expectimax, is able to perfectly model a game 2048. I intended to create an arbetry expectimax tree and calculate the scores of each node. The decision tree required in the expectiminimax algorithm (Figure 1) has four types of nodes. However, only two three of nodes are needed for the problem of solving 2048. These three types of nodes include:

- Terminus nodes: These nodes are the leaves of the tree. Their score is already known and used to calculate the score in the rest of the tree.
- Chance nodes: These nodes represent situations where there are random states that may follow. The weights between a chance node and its children represent the probability of that event occurring. A chance node's score is the weighted sum of its children.
- Maximising nodes: These nodes take the score of the maximum-scoring child.

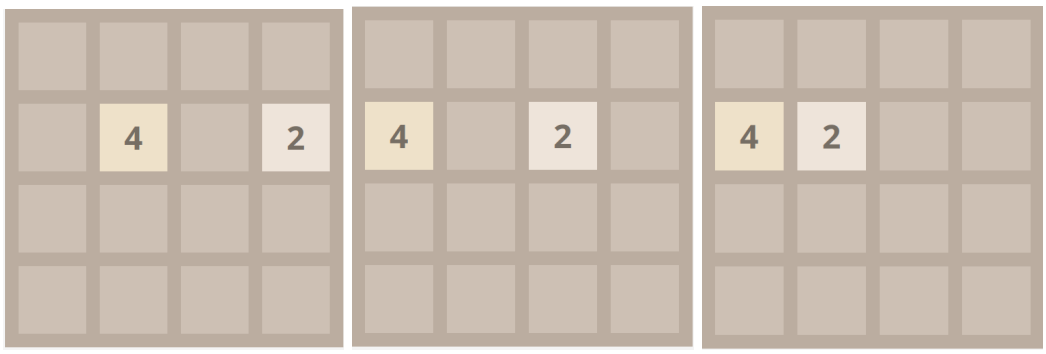

Figure 2: The first image shows a state shown before a left slide, the second one shows after an incorrect left slide, and the third one shows after a correct left slide.

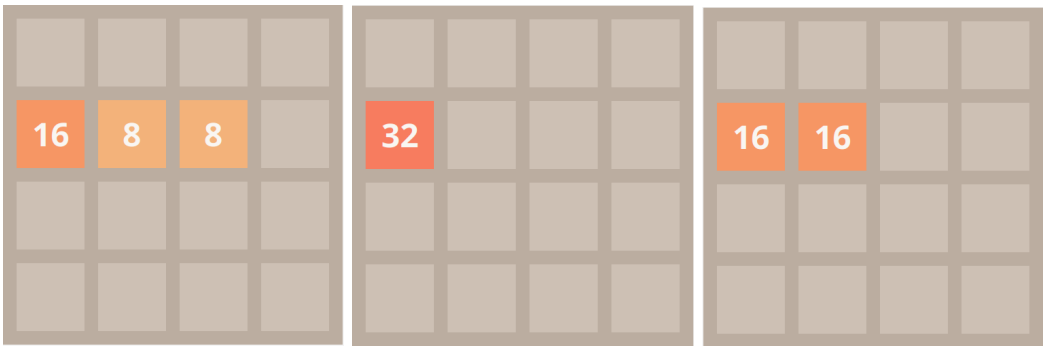

Figure 3: The first image shows a state shown before a left slide, the second one shows after an incorrect left slide, and the third one shows after a correct left slide.

- Minimising nodes: These nodes take the score of the minimum-scoring child.

Each of these three relevant types of nodes has been converted to individual java classes, caching the scores in a float attribute, so they only need to be calculated once.

Figure 1 has

2.3 2048

This proof of concept was created mostly using one source [7], the original 2048 source code. While I did not directly copy any of the code I read through most of the code relevant to the key features of the game and tried to understand why things were done that way. I then re-wrote the code in java using similar approaches.

I learnt a few things that were very useful while writing this prototype. For example:

- If you loop over the numbers in the wrong order a tile may collide with a tile that will be moved later. See Figure 2 for an example.
- When merging tiles, you need to ensure that you do not merge a tile multiple times. See Figure 3 for an example.

My test suite picked up the first issue; however, I had not considered the second case before writing the proof of concept. I only caught the second issue because I wrote a command line user interface (see figure 4 allowing me to play the game and pick up on bugs).

2.4 2048 with Expectimax

A working prototype of expectimax has been developed (see 2.2). A working 2048 prototype has also already been developed as well (see 2.3). These components need to be modified so that the expectimax algorithm can be applied to 2048 game. The aim is to generate the full tree instead of using a heuristic but to do this; the 2048 puzzle must be simplified.

Two modifications were made to the setup of 2048 to ensure the tree was a reasonable size:

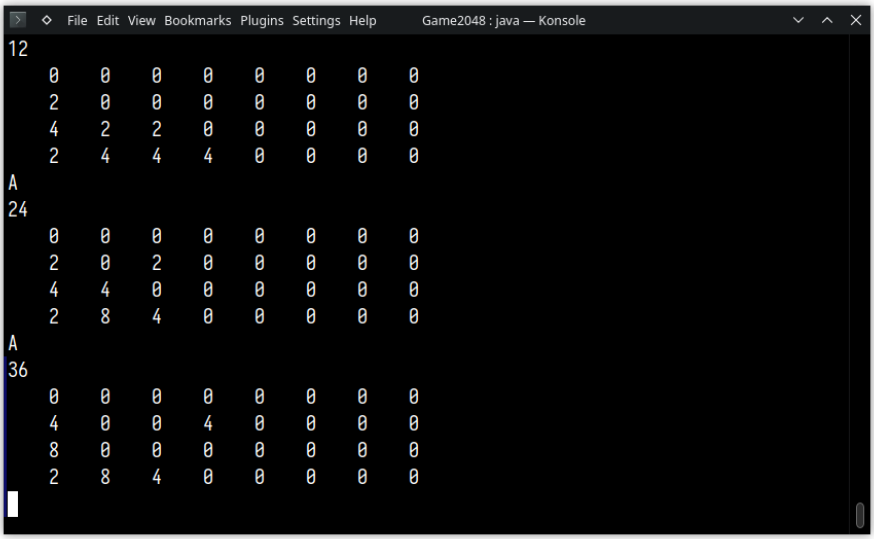

Figure 4: Command line user interface used by 2048 prototype playing a 4 x 8 game.

- Make the game 2×2.
- After a move, only the number 2 can appear.

With these modifications, there are at most only three possible free cells where a tile can be added. With possible moves, the tree can only grow by $3 \times 4 = 12$ for each move. A traditional 2048 game has four possible moves and 15 possible free cells where a tile can be added, meaning the tree can grow by $15 \times 4 = 60$ for each move.

Originally the only modification to the game was making it 2×2 to increase the tree by $6 \times 4 = 24$ for each move. However, the prototype could not calculate the tree within a reasonable time. Hence why the new tiles are limited to just the '2' tile to reduce the complexity further.

2.5 Heuristic for 2048

In this prototype, the restrictions were removed from the game. The size of the game is now $n \times m$, and either a 2 or 4 can appear in the grid. This makes it impracticable to calculate the full tree. To do this, an arbitrary

Two different heuristic functions were implemented.

- Sum up all the items in the grid. This was a simple heuristic to ensure that the algorithm worked as expected. This heuristic could not achieve high scores.
- Sum up all the values multiplied by the row they are in (Top row = 1 and bottom row = n). The most rewarding path was placing larger numbers lower in the grid. This heuristic was much more effective, reaching a tile of 512.

With this, the algorithm can theoretically be applied to any 2048 game, which becomes impractical for a large $depth, n$ or m. The original plan was to apply this to a 2x2 game; however, due to the nature of the heuristics, I removed this requirement and allowed any size of grid, as the $depth$ limit makes this practical.

3 User Interface Design

The user interfaces resemble the original 2048 game, as shown in Figure 5. Parts of the user interface are appropriate for my solver. However, some features must be removed.

Firstly the instructions on how to play the game seen at both the top and bottom of the page have been removed as the game will play itself. Secondly, a new solve button has been added.

The 'best' score box has been removed from the interface. I will need to keep track of more data than the best score to evaluate how effective the algorithm is and will likely log this in a CSV file.

Figure 5: On the left is a screenshot of the official 2048 user interface [2] and on the right is an edited version of this image designed to represent what my main interface might look like

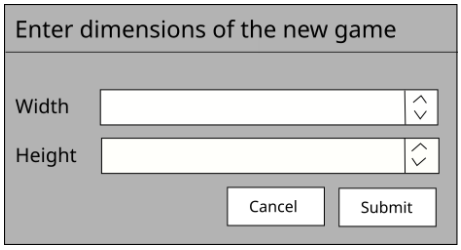

Figure 6: Mock up of the new game popup window.

Most versions of 2048 have an animation that plays as the tiles slide; while this would be possible, as animations do exist in JavaFX [8], I am not familiar with them and think it is not worth the time to re-implementing the animations.

When creating a new game, it is important that the user can enter the size of the new game; however, this input should not be visible when the user doesn't want to make a new game. A long-established solution is providing the user with a popup dialogue asking for the size of the game. There is no reference for this in the original 2048 to base the user interface on, so instead of modelling an existing interface, I have created a mock-up of what the interface should look like figure 6

4 Design Patterns for AI Search

There are many ways design patterns can be used in AI search algorithms, specifically in the expectimax algorithm. Figure 7 shows the structure of the code used for the algorithm; there are a few changes that have been made that have not yet been implemented.

4.1 Factory Design Pattern

A factory design pattern is a creational design pattern used to hide the complexity of creating an instance of a class [9]. This implementation of the expectimax algorithm includes three types of nodes:

- `MaxNode`
- `ChanceNode`

- `LeafNode`

The expectimax tree will always start with a maximising node. This means that despite the complexity of the Node classes, given a game state and the maximum depth of the tree, it is very simple to create the root node of the initial tree. Generating the child nodes is slightly more complex than the initial root node. Depending on the parent and if the node has children, it can be any of the three node types. For this, two creational methods can be created. One is to create an initial root node that only takes in a small amount of information and one that is package private that takes in more data to determine what type of node it is.

4.2 Singleton Design Pattern

A singleton is a creational design pattern that ensures at most one class instance; if an object is required multiple times, the same object is returned each time [9].

The most clear-cut case for this design pattern is the heuristic classes. The existing heuristic classes have no attributes, just two methods. This means that if multiple class instances were created, they would always be equivalent. This means there is no benefit from having multiple objects for these; a Singleton design pattern is a logical choice.

Another case where a singleton pattern could be used is for the Node Factory. Currently, this is just a class containing only static methods. The only situation where two different instances of this factory would be useful is playing multiple 2048 games simultaneously with different random number generators, which seems unlikely.

4.3 Observer Design Pattern

An observer is a behavioural design pattern that is particularly useful when something needs to happen in an object based on an event in another object. A common example of where this design pattern can be used is event handlers in a user interface [10].

An observer interface consists of one function, often called `notify()`, or `update()`. This function is called by an observable object when some event occurs.

In this project, there is a class called `Solver`, which contains the code for the expectimax algorithm. Each time it calculates, and makes, a move, this needs to be updated in the View. To make this happen, an observer design pattern is used. Each time the observer is notified, it includes a parameter that describes the current state of the game, which is passed to the view to be displayed.

4.4 Visitor Design Pattern

Allows some processing to be done externally from the object (A). A second object (B) can visit object A, which accepts object B and finally runs the process on object A. This is particularly useful when there are multiple ways something can be evaluated or multiple things to be evaluated [10].

To be practical, the expectimax algorithm applied to 2048 requires a heuristic function. There are many different heuristic functions, and there may be various reasons to use one over another in certain scenarios; it is convenient to separate the heuristic code from the `GameState` and apply the correct heuristic function when needed.

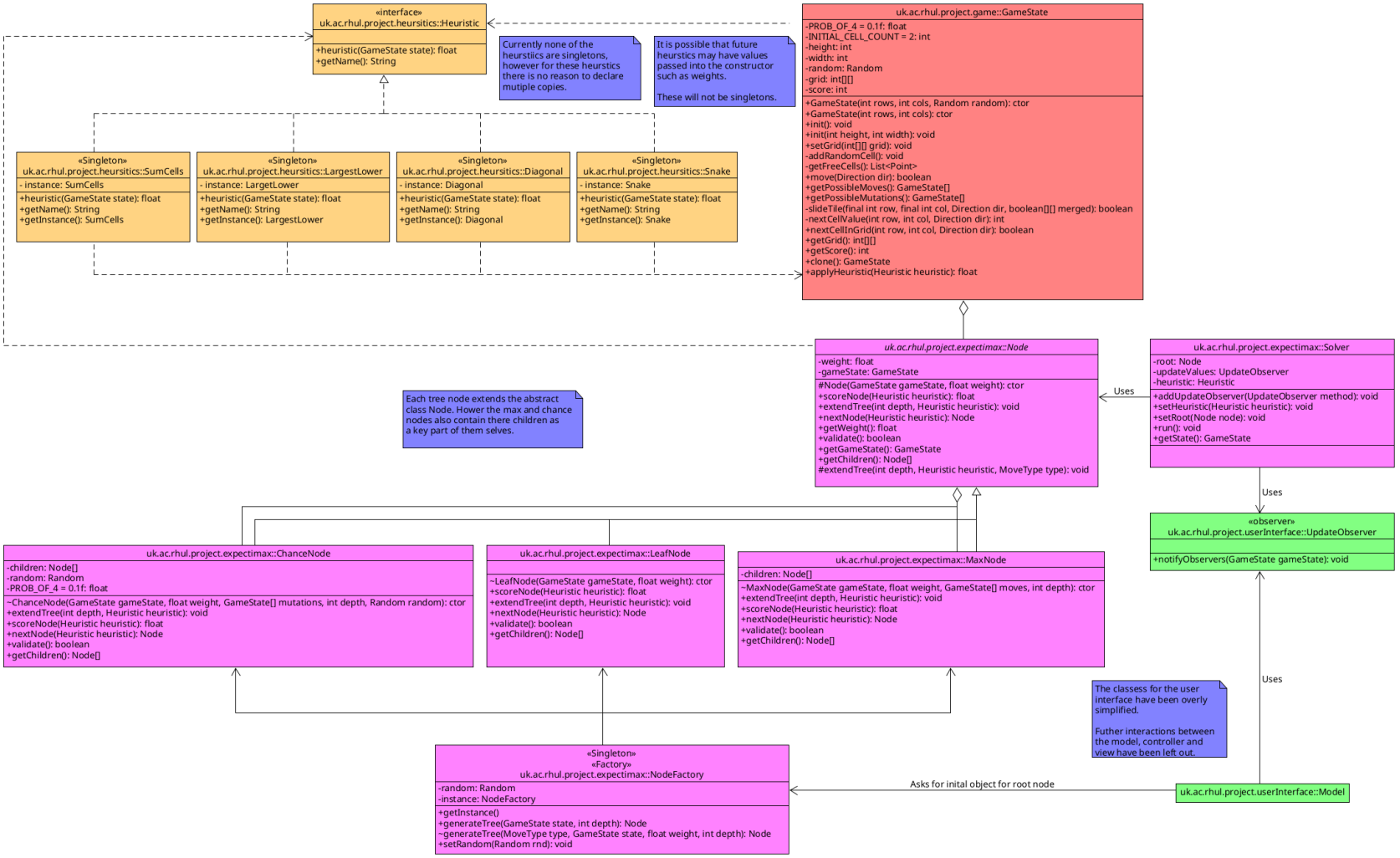

Figure 7: UML Diagram representing the parts of the code related to the expectimax algorithm.

Figure 8: Gradient pattern used in a 2048 human strategy [11].

Figure 9: Example of a trapped value in 2048 [11].

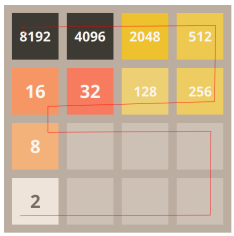

Figure 10: Snake pattern.

5 Techniques used to solve the game

5.1 Human Approaches

It can be difficult to automate a human strategy however it is possible to reward or penalise certain states, and human strategies can be a good place to start when looking for features to reward or penalise.

One human strategy is to keep the largest number in a corner and have a gradient leading to the smaller values like in figure 8 [11]. In testing, this way of arranging the grid makes it relatively easy to merge multiple tiles in a few moves and reduces the risk of trapped values. However, it does lead to many duplicate values that are difficult to merge, so there is often insufficient space to reach the larger tiles.

A trapped value is surrounded by larger numbers [11], such as the 2 in figure 9. Partially when there are not very many free cells around, these cells can be very difficult to get rid of and can take up useful space. They often appear when an empty tile is created after a merge in an area with large numbers.

Another approach is to try and order the tiles into ascending order using a snake shape (Figure 10 [4]. The source for this approach describes an automated method; however, humans can also use this. In testing, this was found to be a very reliable strategy, even reaching tile 8192. The main drawback of this approach was that when merging the row with the largest numbers, there is a risk of producing a trapped value. With careful planning and some luck, it is possible to recover from these states.

5.2 Automated Approaches

Many algorithms can be used to solve 2048, to varying degrees of success. Some of these algorithms are [12]:

- Minimax - assumes that two rational agents always make their most optimal move [13], this strategy does not adapt very well to 2048, according to [12] when citing [14].
- Expectimax - This is a much more effective strategy, similar to Minimax; however, the second agent is assumed to make random decisions [6, p. 200]. This model lends itself much better to 2048 as it exactly matches what happens [5].
- Monte-Carlo Tree-Search - "Produces asymmetric trees, effectively pruning poor paths allowing for deeper searching on paths with greater potential." [12]. This method was highly effective compared to Mini-max and Expectimax.

- Average Depth-Limited Search - "ADLS approximates expectimax by running multiple simulations; it does not try to calculate all possibilities. Instead, likely outcomes will repeat more often." [12]. This method was highly effective.

While there are many different algorithms, some can be very effective; this report will focus on the expectimax algorithm. This is a relatively simple but effective algorithm [5]. As discussed in section 2.4, calculating an expectimax tree for even a small 2048 game is normally impractical; for a traditional 2048 game, there are even more possibilities, leading to an even bigger complete expectimax tree. Previous projects have used a depth-limited expectimax tree [4], sometimes even with a dynamic depth [5].

A heuristic function is needed to evaluate how good a state is at the end of the tree. Two interesting heuristics will be covered in this section.

5.2.1 Snake Heuristic

The snake heuristic is based on the most effective heuristic in [4].

This heuristic works by calculating a weighted sum of the values in the grid:

$$\text{Let } S \text{ be the state of the game, } W = \begin{bmatrix} 4^{15} & 4^{14} & 4^{13} & 4^{12} \\ 4^{8} & 4^{9} & 4^{10} & 4^{11} \\ 4^{7} & 4^{6} & 4^{5} & 4^{4} \\ 4^{0} & 4^{1} & 4^{2} & 4^{3} \end{bmatrix}$$

$$\text{and } h(S) \text{ be the heuristic function.}$$

$$h_1(S) = \sum_{i=1}^{4}\sum_{j=1}^{4} W_{ij}S_{ij}$$

This heuristic takes the human strategy of organising into snake shape rewards boards where this strategy has been applied.

5.2.2 Diagonal Heuristic

The diagonal heuristic is used in the project [5]. Similar to the snake heuristic, part of this is a weighted sum. However, there is also a penalty.

The penalty function $p(i,j) = \sum |\text{difference between each nonzero neighbour}|$. For example, with the grid state

$$\begin{bmatrix} 128 & 64 & 32 & 16 \\ 8 & 32 & - & - \\ 32 & 16 & 8 & - \end{bmatrix}$$

$$P(2,2) = |32-64| + |32-16| + |32-8| = 32+16+24 = 72$$

The heuristic function, where S is the games state, and $W = \begin{bmatrix} 6 & 5 & 4 & 1 \\ 5 & 4 & 1 & 0 \\ 4 & 1 & 0 & -1 \\ 1 & 0 & -1 & -2 \end{bmatrix}$ is:

$$h_2(S) = \sum_{i=1}^{4}\sum_{j=1}^{4} W_{ij}S_{ij}^2 - \sum_{i=1}^{4}\sum_{j=1}^{4} p(i,j)$$

5.2.3 Compassion between Snake and Diagonal Heuristics

After implementing these heuristics, the maximum tiles and scores were logged for 100 games using each heuristic. There was no statistically significant difference between the two heuristics, as shown in Figure 11.

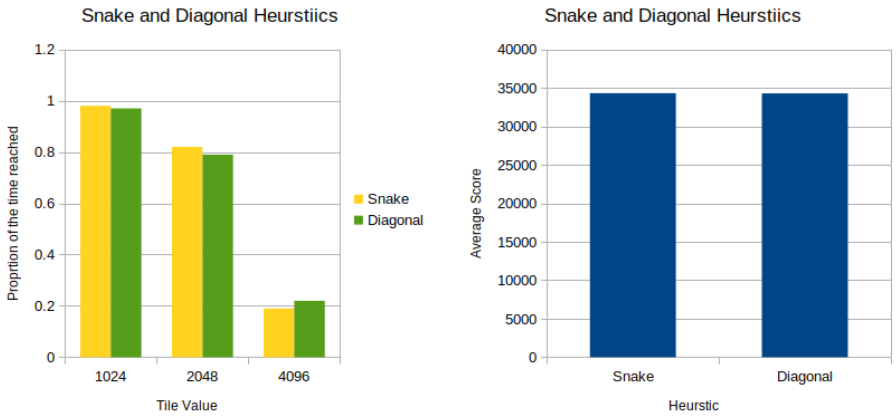

Figure 11: A compassion between the snake and the diagonal heuristics.

5.3 Potential future heuristics

During the second term, I would like to try applying the penalty, defined in section **??** function, to the snake heuristic multiplied by various weights.

Combing the snake and diagonal heuristic. Placing large numbers along one edge is a common strategy in 2048; however, the diagonal heuristic was very effective and quickly got mid range numbers; using the diagonal heuristic to get a mid-range number, such as 128 or 256, but using the snake heuristic on the number tile at the same time. For example, if the game state is.

$$\begin{bmatrix} 2048 & 512 & 128 & 16 \\ - & 4 & 16 & 8 \\ - & - & 4 & - \\ - & - & - & - \end{bmatrix}$$

The weights matrix may look something like this:

$$\begin{bmatrix} 7 & 6 & 5 & 4 \\ 2 & 3 & 4 & 3 \\ 1 & 2 & 3 & 2 \\ 0 & 1 & 2 & 1 \end{bmatrix}$$

6 Time Complexity

6.1 Complexity

6.1.1 4 x 4 Game

As the most effective heuristics are currently locked into a 4×4 game, I will calculate the time complexity when run on a 4×4 game, where n is the maximum depth of the tree.

After each maximising node, the tree grows by at most 4. One for each possible move up, down, left and right.

A game starts with two cells; hypothetically, let's say:

$$\begin{bmatrix} 2 & 2 & - & - \\ - & - & - & - \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

The best case outcome is that these two cells are merged:

$$\begin{bmatrix} 4 & - & - & - \\ - & - & - & - \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

This leaves 15 free cells for the new cell to appear; after this, there will be two cells again. Therefore there can never be more than 15 free cells.

Therefore the most children a chance node can have is $15 \times 2 = 30$.

Each node has constant time work, so the leaf nodes dominate the tree.

For every two layers, the number of leaf nodes multiplies at most 120, and there are n layers in the tree. Therefore their are $120^{\frac{n}{2}}$ leaf nodes $\therefore$ time complexity to calculate the next move is $120^{\frac{n}{2}} = O(\sqrt{120}^{n}) \approx O(10.954^n)$

6.1.2 m x n Game

Consider calculating the children of a max node: each max node has at most 4 children, and each child's state is calculated using the following code (see uk.ac.rhul.project.game.GameState):

```
boolean move(Direction dir)
{
    ...
    for (int i : dir.getVerticalStream(this.height))
    {
        for (int j: dir.getHorizontalStream(this.width))
        {
            if (grid[i][j] != 0 && this.slideTile(i, j, dir, merged)) ...
        }
    }
    return flag;
}

private boolean slideTile(final int row, final int col, Direction dir, boolean[][] merged)
{
    int target_row = row;
    int target_col = col;

    // Calculate how far the tile can be moved (assuming no merge)
    while (nextCellInGrid(target_row, target_col, dir) && this.nextCellValue(target_row, target_co
    {
        target_row += dir.getRows();
        target_col += dir.getCols();
    }

    ...
}
```

$\Rightarrow$ For directions UP and DOWN, the time complexity of the move operation is $O(m^2n)$ and for LEFT and RIGHT the time complexity is $O(mn^2)$

$\Rightarrow$ calculating the children of the max node has time complexity $2O(mn^2) + 2O(m^2n)$.

Consider calculating the children of a chance node:

A grid has at least one nonzero tile $\therefore$ There are $mn - 1$ cases to consider.

By looking at Figure 12, it can be deduced that the number of work increases after every pair of layers, and the amount of work done generate the last two layers $d - 1$ and d is:

$$\begin{aligned}
\text{Time Complexity} &= 4^{\frac{d}{2}}(mn-1)^{\frac{d-2}{2}}O(mn(m+n)) + 4^{\frac{d}{2}}(mn-1)^{\frac{d}{2}}O(mn)) && (1)\\
&= O(4^{\frac{d}{2}}(mn-1)^{\frac{d-2}{2}}mn(m+n) + 4^{\frac{d}{2}}(mn-1)^{\frac{d}{2}}mn) && (2)\\
&= O(4^{\frac{d}{2}}(mn-1)^{\frac{d-2}{2}}mn(m+n+\underline{mn}-1) && (3)\\
&= O(4^{\frac{d}{2}}(mn-1)^{\frac{d}{2}}m^2n^2) && (4)
\end{aligned}$$

When scoring the tree, all nodes, apart from the leaf nodes, are constant time, the leaf nodes are

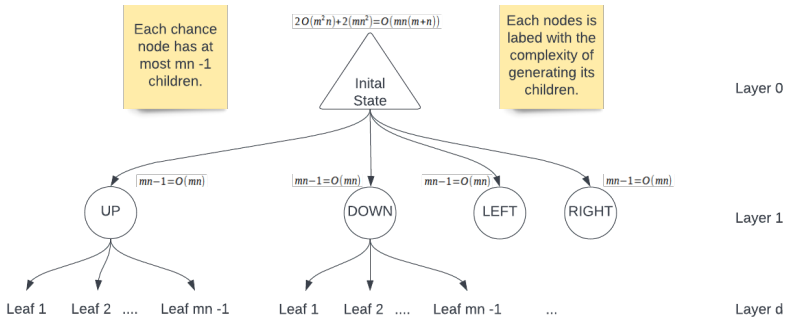

Figure 12: A tree for a nm 2048 game

$O(nm)$ ∴, scoring the tree is $O(4^{\frac{d}{2}}(mn-1)^{\frac{d}{2}}nm)$. The term for generating the tree is still more significant, so the time complexity $= O(4^{\frac{d}{2}}(mn-1)^{\frac{d}{2}}m^2n^2)$.

7 Professional Issues: Licensing

Licensing is relevant to this project as the game 2048's source code is available under the MIT Licence on GitHub [7]. This is a very permissive licence, meaning there are few restrictions on how one can use the published work for [15]. This allows for ethical dubious situations where a large company uses the code internally without publishing, paying for or even acknowledging the use of that code. To deal with this issue copyleft license were created (see 7.1).

7.1 Open-source Software

Open-source software is any software released under a license considered to be open-source. These licenses allow the source to be "freely accessed, used, changed and shared (in modified or unmodified form) by anyone." [15].

There are two major categories of open-source licenses: copyleft and permissive licenses.

A copyleft licence, such as the various versions of the GNU General Public License [16]. The main purpose of copyleft is to protect open-source software from being converted to or re-used in property software. Copyleft licenses tend to be more complex and difficult to understand; for example, GPLv3 is 674 lines, while the permissive MIT license only has 21 lines. Complex licences can lead to issues, such as licence incompatibility; for example, the CDDL license is widely considered to be incompatible with the GLP licence [17]; this makes it difficult to use code from software from these two licenses in the same code-base, despite the similar intent of these two licences. The simplest definition of a permissive license is simply an open source licence that is not copyleft [15].

7.2 Open source software in this project

This project has benefited from various pieces of open software, more than is possible to credit. Some examples are:

Environment:

- OpenJDK (GPL-v2.0)
- Maven (Apache 2.0)
- Git (GPL-v2.0)
- Overleaf (AGPL-3.0)
- Intellij (Apache-2.0)

Refrences:

- 2048 (MIT)

Video recording/editing software:

- Kdenlive (GPLv3)
- OBS studio (GPL-2.0)

Libraries:

- JUnit Jupiter API (EPL 2.0)
- JavaFX (GPL-v2.0)
- Jackson Data format CSV (Apache 2.0)

Despite the fact that this project has massively benefited from the software listed above, there is no evidence of the Video recording/editor software or some of the software under environment. It is not fair on these projects that they don't get the credit they deserves. For example the video editor Kdenlive (Made by the KDE Team), was used to place the clips of the program running in the video with text and music.

Some of the source code used in the project is simply added from the original MIT 2048 code [2], however being a permissive license, this imposes no restrictions on how the how can be used in the future. The other software listed above has only been used in the development, or is required to run the software. For example, the code could be packaged into a binary and sold with no mention of the open source software without breaking the licenses. While legally this behaviour would be consider acceptable, from an ethical perspective this would be highly unethical as it would be profiting of another developers work without even crediting that developers work.

Currently no source code has been used under a copyleft license in this project, however if at this were the case, a situation with a binary being sold would still be possible. The difference would be that the source code would have to be published under a compatible license, and freely available. While it would still be profiting with the aid of another developers code, their are other options that do not involve paying for the software, such as compiling from source, or downloading a third party binary from another source. This means that, for a well researched user, the payment for a binary is a choice rather than the only way of using the software.

While it is not required by the license, if this code was every published, it would certainly be worth putting a link to the 2048 GitHub page [2], as it was an inspiration to work on the project. Thought this project [2] has been described as the source code for 2048, however in the README file Gabriele Cirulli clearly states that the project is a clone of the Android app 1024, and Saming's 2048. While the source code for neither of these is public, and Saming's 2048 is no longer available, I have been incorrectly implying that the concept of 2048 was created by Gabriele Cirulli.

7.3 Unlicensed Source Code and Copyright

Many publicly available GitHub project, partially small ones, have never been licensed. In many of these situations it is likely that the original developer does not object to people using their code, as they have been through the work of publishing it in the first place. Technically, using code from project like this is in breach of the owners copyright. GitHub makes it very easy to accidentally make the mistake of including unlicensed code, or even basing an entire project on it. For example, even if the licence on a project does not allow for modification of the project, a fork button is displayed to the user. A fork can be created without the user even being warned.

The heuristic function, diagonal's (see 5.2.2 code is currently heavily based on the code from the unlicensed GitHub project 2048-Game-Using-Expectimax [5], as this project it covered by the educational exception to copyright, this is not a problem. From an ethical perspective, as long as the code remains publicly available and isn't used commercially, there is unlikely to be an issue, however it does leave the project vulnerable to copyright claims if it is published. To avoid the risk of copyright claims either the unlicensed code must be removed from the project, or permission must be obtained to use the code.

References

[1] "Demonstration of code." [Online]. Available: https://drive.google.com/file/d/11TPyogpjW0ArU3878IWw9uVNiqG5jj_y/view?usp=share_link

[2] C. Gabrielle, "Original 2048 game," 2014. [Online]. Available: https://gabrielecirulli.github.io/2048/

[3] 2014.

[4] N. Yun, H. Wenqi, and A. Yicheng, "Ai plays 2048." [Online]. Available: http://cs229.stanford.edu/proj2016/report/NieHouAn-AIPlays2048-report.pdf

[5] W. Syed Abdul, B. Muhammad, and D. Hamza Javed, "2048 game using expectimax," 2018. [Online]. Available: https://github.com/Wahab16/2048-Game-Using-Expectimax

[6] S. Russell, S. Russell, P. Norvig, and E. Davis, *Artificial Intelligence: A Modern Approach*, ser. Prentice Hall series in artificial intelligence. Prentice Hall, 2010. [Online]. Available: https://books.google.co.uk/books?id=8jZBksh-bUMC

[7] C. Gabrielle, "Original 2048 source code," 2014. [Online]. Available: https://github.com/gabrielecirulli/2048

[8] *OpenJFX 19 API Documentation*. [Online]. Available: https://openjfx.io/javadoc/19/

[9] D. Cohen, "Creational design patterns," 2021.

[10] ——, "Behavioural design patterns," 2021.

[11] "2048 strategy," 2022. [Online]. Available: https://www.2048.org/strategy-tips/

[12] P. Rodgers and J. Levine, "An investigation into 2048 ai strategies," in *2014 IEEE Conference on Computational Intelligence and Games*, 2014, pp. 1–2.

[13] L. Zhang, "Cs2910 - adversarial search," 2021.

[14] "Mini-max ai," 2014, source is not available anymore, cited by [12]. [Online]. Available: http://ov3y.github.io/2014-AI

[15] "Open source initiative faq." [Online]. Available: https://opensource.org/faq

[16] "What is copyleft?" [Online]. Available: https://www.gnu.org/licenses/licenses.html#WhatIsCopyleft

[17] "Common development and distribution license (cddl)." [Online]. Available: https://www.gnu.org/licenses/license-list.en.html#CDDL