Solving 2048 with Expectimax

David Cook

March 14, 2023

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 does not have a lot of accompanying documentation. It is difficult to understand what is happening.

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.

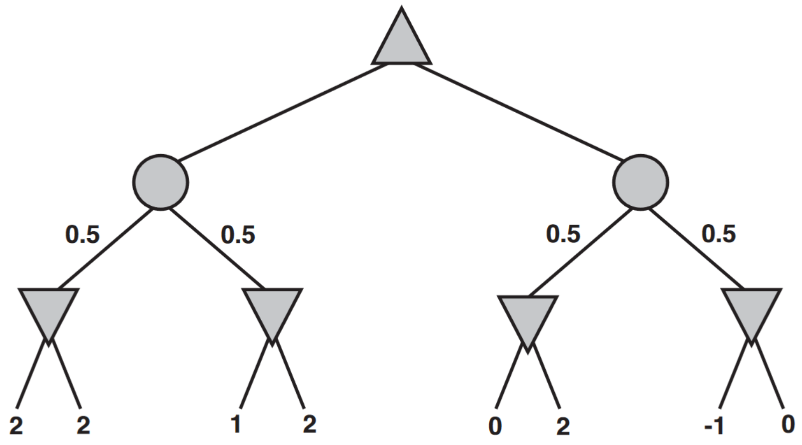


Figure 1: An expectiminimax tree example [6, p. 200]

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, three 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.

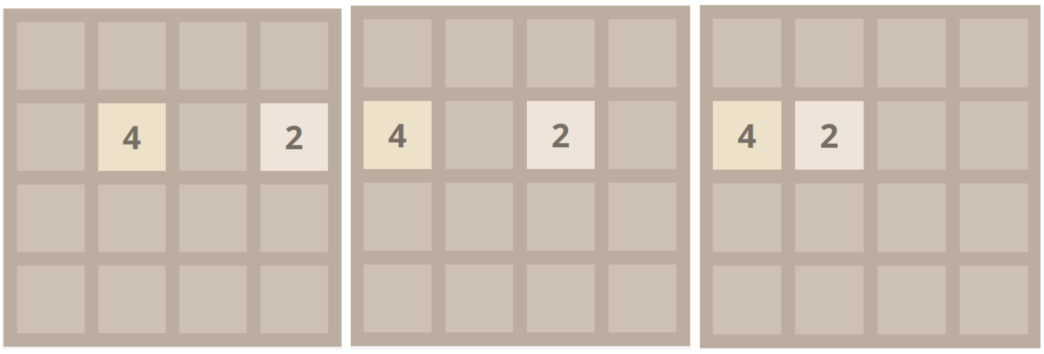


Figure 2: The first image shows a state shown before a left slide, the second one shows after an incorrect left slide, and the third one shows after a correct left slide.

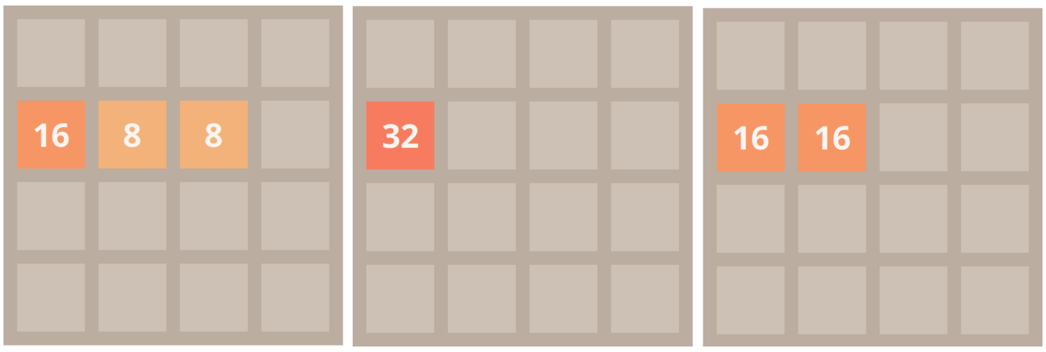


Figure 3: The first image shows a state shown before a left slide, the second one shows after an incorrect left slide, and the third one shows after a correct left slide.

- 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:

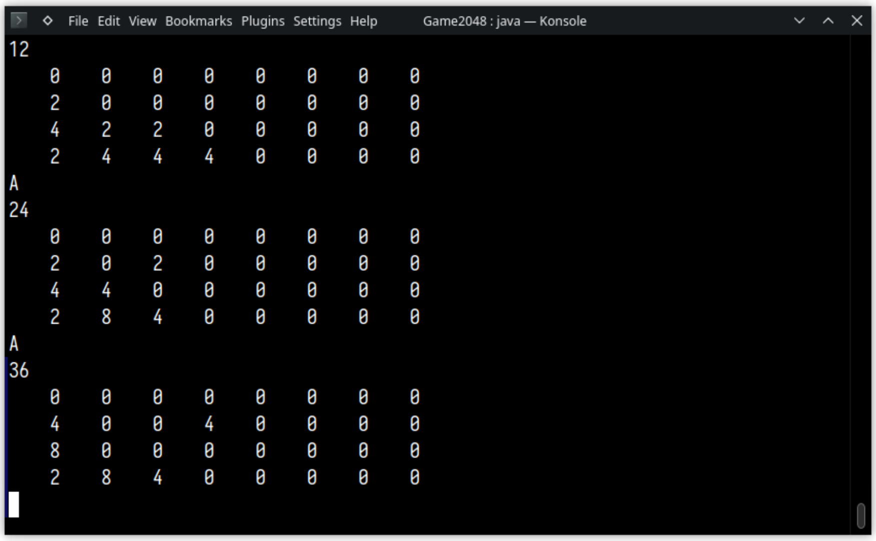
- If you loop over the numbers in the wrong order, a tile may collide with a tile that will be moved later. See Figure 2 for an example.
- When merging tiles, you need to ensure that you do not merge a tile multiple times. See Figure 3 for an example.

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 already been developed (see 2.3). These components need to be modified to apply the expectimax algorithm to the 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:


File Edit View Bookmarks Plugins Settings Help
Game2048 : java — Konsole

```
12
    0    0    0    0    0    0    0    0
    2    0    0    0    0    0    0    0
    4    2    2    0    0    0    0    0
    2    4    4    4    0    0    0    0
A
24
    0    0    0    0    0    0    0    0
    2    0    2    0    0    0    0    0
    4    4    0    0    0    0    0    0
    2    8    4    0    0    0    0    0
A
36
    0    0    0    0    0    0    0    0
    4    0    0    4    0    0    0    0
    8    0    0    0    0    0    0    0
    2    8    4    0    0    0    0    0
```

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, the new tiles are limited to just the '2' tile to further reduce the complexity.

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 the 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.

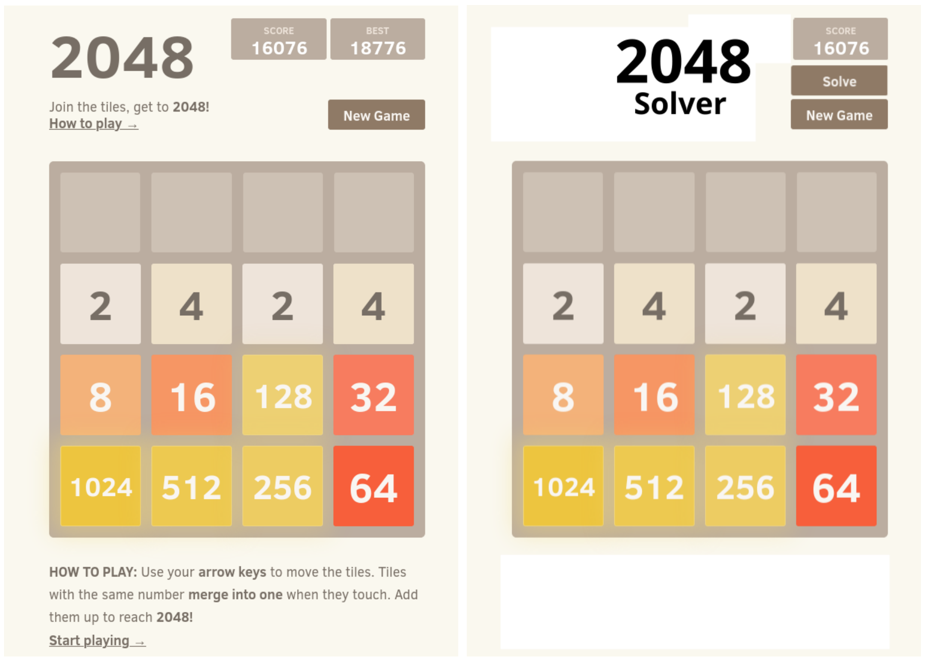

2048
SCORE
16076
BEST
18776
Join the tiles, get to 2048!
How to play →
New Game
2 4 2 4
8 16 128 32
1024 512 256 64
HOW TO PLAY: Use your arrow keys to move the tiles. Tiles with the same number merge into one when they touch. Add them up to reach 2048!
Start playing →
2048
Solver
SCORE
16076
Solve
New Game
2 4 2 4
8 16 128 32
1024 512 256 64

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

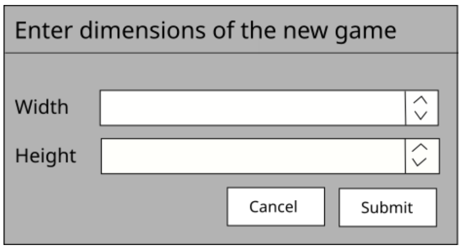


Figure 6: Mock up of the new game popup window.

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]. The original 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.

Since implementing multi-threading, the process of making nodes is slightly simpler. There is one node class `Node`. Its behaviour comes from a class implementing the `NodeBehaviour` interface. See section 4.2 for more about this.

Another potential use case for the Factory design pattern would be making a class to generate the correct views from a given set of command line arguments. Previously the Driver class' main method was used as a proto-factory for the views. Based on the inputs, this function determined which view to create.

4.2 State Design Pattern

A state is a behavioural design pattern used when an object's behaviour needs to change based on some condition [10]. A class using that state design pattern has an attribute that will be referred to as `objectState`. `state` will have the type of some interface or superclass. When the 'state' of the object is changed, the `objectState` is changed to a different implementation of the interface. Any features of the class that depend on the state are delegated the concrete implementations of this interface.

The state design pattern is used in the `Node` class. Each node has a behaviour which manages what happens when a heuristic is applied to the node, when the next node is requested and manages the node's children. Every node starts as a `LeafNodeBehaviour` whih has the following features:

- Stores no data related to children; it is known there aren't any.
- `nextNode (Heuristic)` simply throws EndOfGameException. This relies on the assumption that the tree has a depth of at least two.
- `applyHeurstic(Heuristic)` applies the heuristic to the games state.

There are two other behaviours `NodeBehaviourMaximize NodeBehaviourChance`. Each stores child nodes in an array, applying the heuristic to their child nodes, taking the most significant or weighted average value. The next node method will either make the best move or pick a random one.

When `generateChildren(...)` is called on a node, an appropriate Node Behaviour will be generated (based on a method passed to the constructor), and the node 'state' will be switched to this new behaviour.

4.3 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.

4.4 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.

4.5 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 uLeafNodefseful 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.

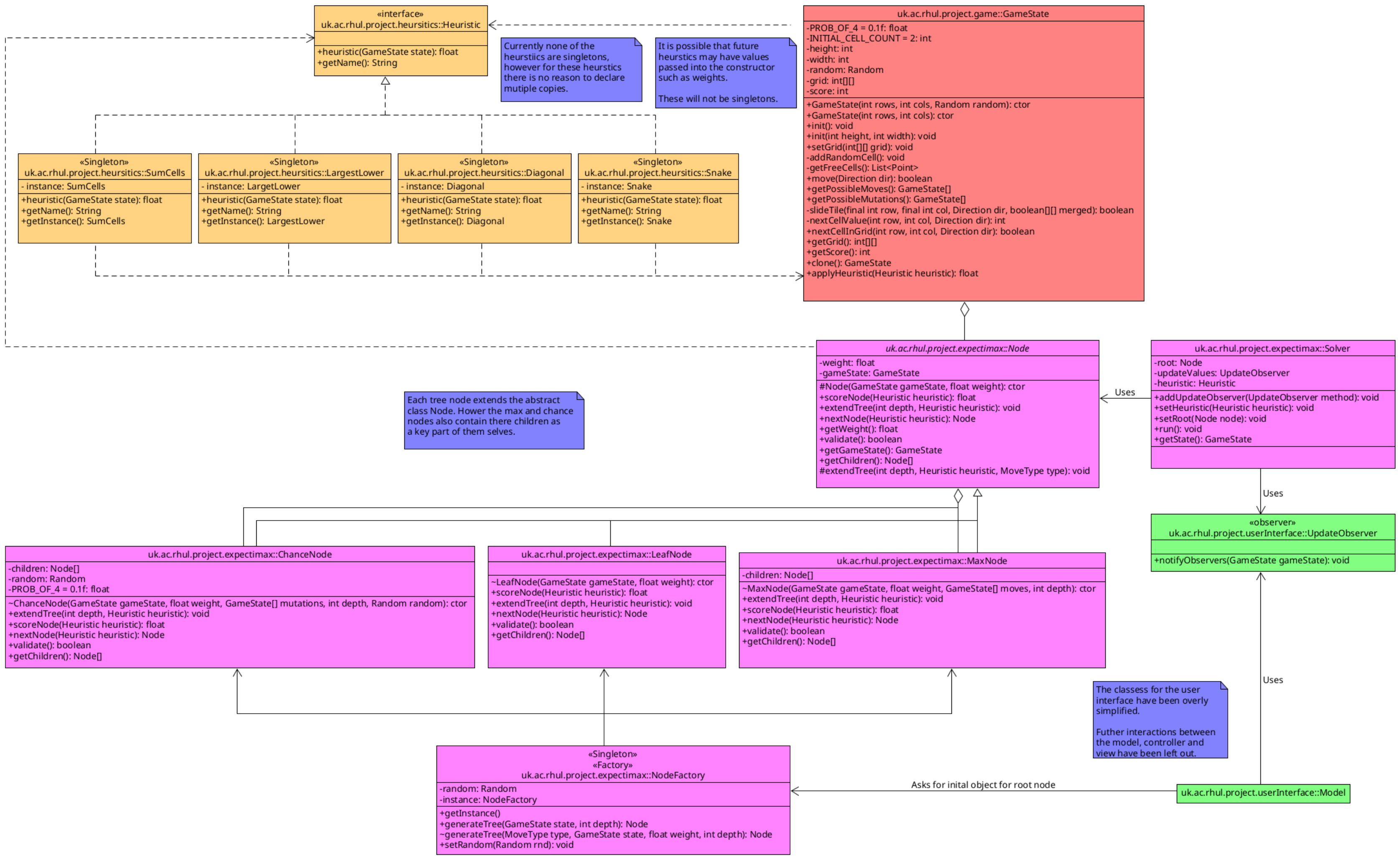

«interface»
uk.ac.rhul.project.heursitics::Heuristic
+heuristic(GameState state): float
+getName(): String
Currently none of the heurstiics are singletons, however for these heurstics there is no reason to declare mutiple copies.
It is possible that future heurstics may have values passed into the constructor such as weights.
These will not be singletons.
uk.ac.rhul.project.game::GameState
-PROB_OF_4 = 0.1f: float
-INITIAL_CELL_COUNT = 2: int
-height: int
-width: int
-random: Random
-grid: int[][]
-score: int
+GameState(int rows, int cols, Random random): ctor
+GameState(int rows, int cols): ctor
+init(): void
+init(int height, int width): void
+setGrid(int[][] grid): void
-addRandomCell(): void
-getFreeCells(): List<Point>
+move(Direction dir): boolean
+getPossibleMoves(): GameState[]
+getPossibleMutations(): GameState[]
-slideTile(final int row, final int col, Direction dir, boolean[][] merged): boolean
-nextCellValue(int row, int col, Direction dir): int
+nextCellInGrid(int row, int col, Direction dir): boolean
+getGrid(): int[][]
+getScore(): int
+clone(): GameState
+applyHeuristic(Heuristic heuristic): float
«Singleton»
uk.ac.rhul.project.heursitics::SumCells
- instance: SumCells
+heuristic(GameState state): float
+getName(): String
+getInstance(): SumCells
«Singleton»
uk.ac.rhul.project.heursitics::LargestLower
- instance: LargetLower
+heuristic(GameState state): float
+getName(): String
+getInstance(): LargestLower
«Singleton»
uk.ac.rhul.project.heursitics::Diagonal
- instance: Diagonal
+heuristic(GameState state): float
+getName(): String
+getInstance(): Diagonal
«Singleton»
uk.ac.rhul.project.heursitics::Snake
- instance: Snake
+heuristic(GameState state): float
+getName(): String
+getInstance(): Snake
uk.ac.rhul.project.expectimax::Node
-weight: float
-gameState: GameState
#Node(GameState gameState, float weight): ctor
+scoreNode(Heuristic heuristic): float
+extendTree(int depth, Heuristic heuristic): void
+nextNode(Heuristic heuristic): Node
+getWeight(): float
+validate(): boolean
+getGameState(): GameState
+getChildren(): Node[]
#extendTree(int depth, Heuristic heuristic, MoveType type): void
Uses
uk.ac.rhul.project.expectimax::Solver
-root: Node
-updateValues: UpdateObserver
-heuristic: Heuristic
+addUpdateObserver(UpdateObserver method): void
+setHeuristic(Heuristic heuristic): void
+setRoot(Node node): void
+run(): void
+getState(): GameState
Each tree node extends the abstract class Node. Hower the max and chance nodes also contain there children as a key part of them selves.
Uses
«observer»
uk.ac.rhul.project.userInterface::UpdateObserver
+notifyObservers(GameState gameState): void
uk.ac.rhul.project.expectimax::ChanceNode
-children: Node[]
-random: Random
-PROB_OF_4 = 0.1f: float
~ChanceNode(GameState gameState, float weight, GameState[] mutations, int depth, Random random): ctor
+extendTree(int depth, Heuristic heuristic): void
+scoreNode(Heuristic heuristic): float
+nextNode(Heuristic heuristic): Node
+validate(): boolean
+getChildren(): Node[]
uk.ac.rhul.project.expectimax::LeafNode
~LeafNode(GameState gameState, float weight): ctor
+scoreNode(Heuristic heuristic): float
+extendTree(int depth, Heuristic heuristic): void
+nextNode(Heuristic heuristic): Node
+validate(): boolean
+getChildren(): Node[]
uk.ac.rhul.project.expectimax::MaxNode
-children: Node[]
~MaxNode(GameState gameState, float weight, GameState[] moves, int depth): ctor
+extendTree(int depth, Heuristic heuristic): void
+scoreNode(Heuristic heuristic): float
+nextNode(Heuristic heuristic): Node
+validate(): boolean
+getChildren(): Node[]
The classess for the user interface have been overly simplified.
Futher interactions between the model, controller and view have been left out.
Uses
«Singleton»
«Factory»
uk.ac.rhul.project.expectimax::NodeFactory
-random: Random
-instance: NodeFactory
+getInstance()
+generateTree(GameState state, int depth): Node
~generateTree(MoveType type, GameState state, float weight, int depth): Node
+setRandom(Random rnd): void
Asks for inital object for root node
uk.ac.rhul.project.userInterface::Model

Figure 7: [[UPDATE DIAGRAM]].

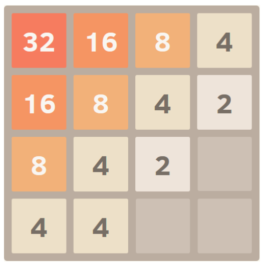


Figure 8: Gradient pattern used in a 2048 human strategy [11].

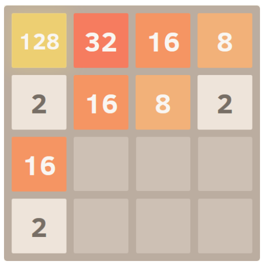


Figure 9: Example of a trapped value in 2048 [11].

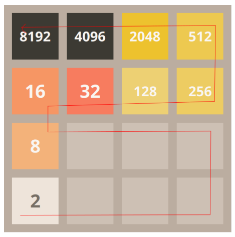


Figure 10: Snake pattern.

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:

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

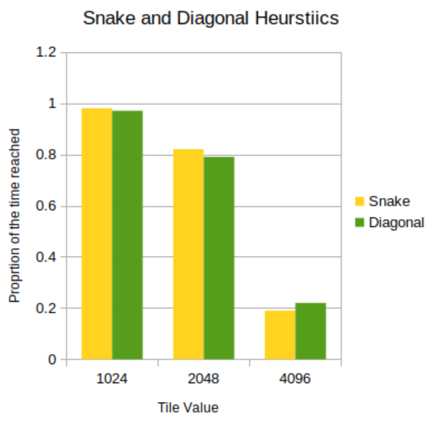
$$P(2,2) = |32-64| + |32-16| + |32-8| = 32 + 16 + 24 = 72$$

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

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

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.



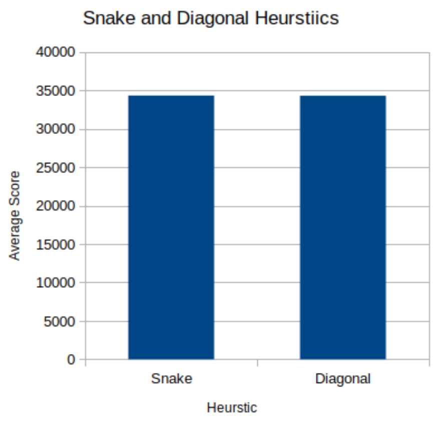


Figure 11: A compassion between the snake and the diagonal heuristics.

6 Dynamically Generating Heuristics for a n x m game

Specific heuristics are easy to adapt to any size of the 2048 game, such as summing up all the cells in a grid; however, generating a weighted matrix for any game is much more difficult. Some heuristics do not have an obvious formula to generate the weights matrix. To start this process, I made variable-sized but still square heuristics.

6.1 Dynamic Snake

As it leads to a simpler algorithm to generate the matrix, the smallest cell it at the top left, and the largest cell will be at the bottom corner (depending on the matrix size). It can be generated using the following code. While the algorithm works on rectangular games, this heuristic is optimised

Algorithm 1 Dynamic Snake

```
1: procedure DynamicSnake(row, col)
2:     powers ← double[row][col]
3:     for $i$ ← 0 to row−1 do
4:         for $j$ ← 0 to col−1 do
5:             if i % 2 == 0 then
6:                 $k \leftarrow j$
7:             else
8:                 $k \leftarrow$ row$-j-1$
9:             end if
10:            power[$i$][$j$] ← $4^{i*col+k}$
11:        end for
12:    end for
13: end procedure
```

for square games. A square game is more symmetrical than a rectangular game. There are eight theoretically equivalent rotations and reflections for a square game. In rectangular games, there would only be four rotations that would be more optimal. To determine which rotations were the best for different, the Dynamic Snake heuristic was run on several $n \times m$ and $m \times n$. In every pair, the heuristic performed better in the games which were wider than tall.

A solution is generating a matrix that goes up from the bottom right along the longer edge, depending on which way that is.

A few techniques were tried to improve the performance of the Dynamic Snake heuristic:

6.1.1 Symmetrical Heuristic

A version of this heuristic was attempted with eight weight matrices, two configurations for each corner. When the heuristic is evaluated, the maximum weighted sum is taken. The expected effect was a slight increase in performance for the heuristic due to the increased flexibility. This was not the result; a significant performance decrease was noted.

6.1.2 Anti-trap

The smallest weighted cell on the bottom row (bottom right in a 4×4 game), from know on this will be refereed to as the trap cell. The cell above the trap cell is only rewarded if its value is less than or equal to the value of the trap cell. This effect was more interesting than that of the Symmetrical change. The percentage of games reaching 8192 was slightly increased, however smaller values were less likely to be reached, the game no longer even reached 2048 in every game.

6.1.3 Fail Wrappers

Fail Ratio is a second heuristic, designed to be wrapped around a different heuristic. If the game would loose in this state, it multiplies the heuristic score be a constant multiplier. It was run with rations -1 $\rightarrow$ 1 stepping with increments of 0.2. The best ratio was 0.6.

Fail Setter was the similar approach, made for a heuristic not mentioned yet. In the event that it game would lose, in this state, the value is set to a specified value. While this led to slight improvement in performance, it was not as effective as the fail ratio.

6.2 Monotonic

The second heuristic designed in a way, where very few adoptions would be needed for rectangular games. It was based on the heuristic used by [3]. There are several key features in this heuristic:

- Values - on the outside edges are added up.
- Monotony - penalises states were row/columns don't increase in the same direction.
- Free Cells - rewards states with more free cells.
- Fail Setter - The values are set to a given value if the state loses, so far, the most optimal value is -1000.

This heuristic also approximates a more effective version of the diagonal and has constantly been scoring the highest in most game states, including rectangular game states. It is flexible enough not to need modifications for specific game states.

There was one modification, which attempted to be more accurate to the original. This would mean each row could increase in different directions. For example:

$$\begin{bmatrix} - & - & - \\ 2 & 4 & 8 \\ 64 & 32 & 16 \end{bmatrix}$$

It would be a relatively high-scoring game state (for monotony) in the original. Despite the discrepancy between the original and my implementation, the approximates

6.3 Preformance

7 Complexity

7.1 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 $120^{\frac{n}{2}} = O(\sqrt{120}^n) \approx O(10.954^n)$

7.2 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:

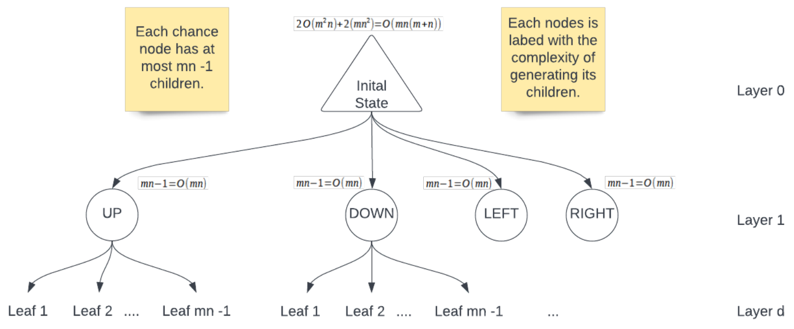


Figure 12: A tree for a n x m 2048 game

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

When scoring the tree, all nodes, apart from the leaf nodes, are constant time, the leaf nodes are $O(nm)$ $\therefore$, 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 $= O(4^{\frac{d}{2}}(mn-1)^{\frac{d}{2}}m^2n^2)$.

8 Optimisations

8.1 Multi-threading

Given the size of a tree for 2048, performance will always be a limiting factor in calculating the game more accurately, in principle the expectimax algorithm is straightforward to multi-thread, ignoring the possibility of pruning; every node is only dependent on its children, not its siblings or parents. It, therefore, follows that each child can be calculated in parallel.

Initially, an attempt was made using the java thread class to process the children in parallel. However, a solution was already built into java. Parallel Streams allow classic stream methods, such as map and foreach to process each element in parallel instead of sequentially [15]. In the process of developing this, a bug was discovered with calculating the depth of the tree; an opportunity was also taken to rewrite how nodes work, reducing the amount of code repetition. This makes it difficult to compare the newest implementation with the older ones. However, an early prototype was compared to the original. When running 100 traditional games with each heuristic function implemented at the time (Sum Cells, Largest Lower, Snake (4x4) and Diagonal (4x4)) using the serial code, it took 18 hours 24 minutes to run. In contrast, on the same machine (4 cores / 8 threads), it took 9 hours and 20 minutes; this is almost double the performance. When monitoring the CPU usage, and almost nothing else is running, there are now very few monuments when a single thread goes below 95% used for a default-size game.

8.2 Pruning

There are two approaches I intend to try for pruning an expectimax tree. 1. Pruning unlikely possibilities. Branches that have higher numbers of 4s can be pruned as they are unlikely to occur.

2. If consecutive bad moves are taken it's unlikely that they can be recovered from. There is less need to explore negative moves compared to positive ones.

8.2.1 Pruning Unlikely Possibilities

This approach to pruning is very straightforward to implement. I added a parameter to the method, which generates the children of a node, count4. On the first call of this method, it should contain

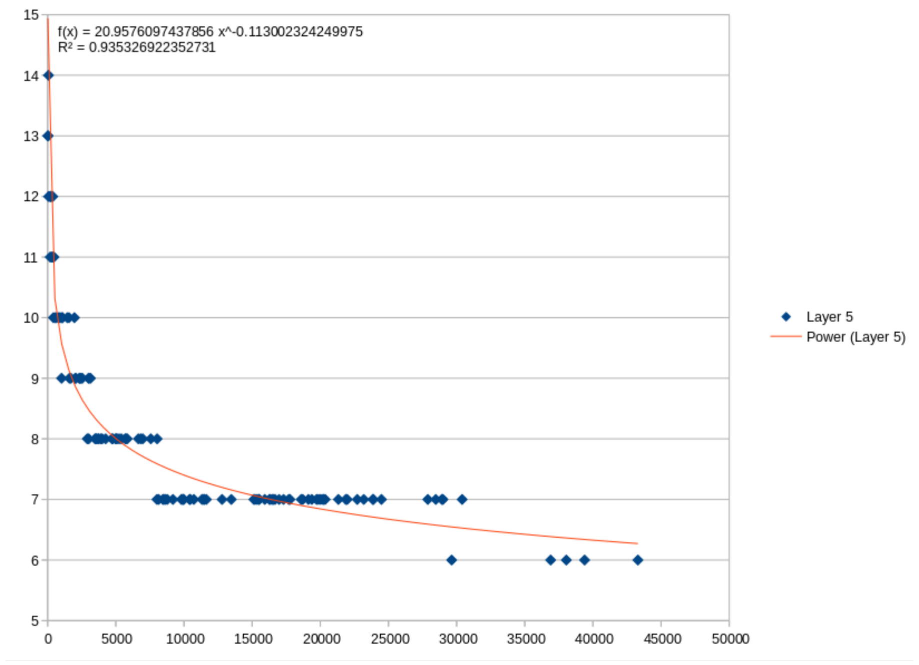


Figure 13: Caption

the number of 4s the approach can contain before values are pruned. For example, if count4=1, when a node adding a four is generated, it will not generate that node's children.

To increase the depth of a tree to 8, all child nodes four must be pruned. However, this performs worse than the regular depth of 7. The pruning has also been tested on dynamic depth games. For example, with the game seed zero, pruning a game allows it to be completed in less than an hour, however, scores less than a game with a constant depth of 7. It is unlikely I will continue to use this feature.

8.2.2 Pruning Poor Moves

The first difficulty here is defining what a poor move is. There are a few approaches that were tested

- Moves which score a significantly lower heuristic than parent nodes even after a few moves.
- Moves that are currently scoring worse than already calculated siblings.

The first approach is much simpler to implement in the current setup. The project relies on multi-threading, so there is a sequence to when they are calculated.

8.3 Dynamic Depth

When there are fewer free cells in the grid, the tree can be explored deeper. As the branching factor on the tree is lower. Creating a function $d(x)$ was a complex process. I started by including a simple feature in the program when the program is started with the command line argument '-d', it is able to probe the tree to see how far be generated in a given period. Using a python script, I picked 10 random game states with 0-14 free cells from a game I had saved to my laptop. I created a CSV file which contained the number of nodes in each layer of the tree, the number of free cells in the state and the maximum depth achieved in 3 seconds.

As I can always process depth 7 on a tree, this means after each moves a tree of depth, 5 is already available, except on the first move. When plotted on a scatter graph, there was quite a clear trend between the number of nodes on the 5th layer and how deep the tree can be processed, see Figure 13.

The function $f(x)$ overestimates some points. I started by instead computing floor($f(x)$); however, this is still overestimated. Over-estimates were common on higher values; to solve this, the power was updated to $h(x) = \text{floor}(max(21x^{-0.065} - 3.75, 7))$.

This test has been very difficult to test, as games are now taking over 1hr. To get a hint of how well this was working, I picked a heuristic (Monotonic with the score set to -1000) when it failed. Using this heuristic, a seed of 0, and a depth of 6 and 7, the complete tree had already been computed (These files have not been included as they total 91.8 GiB). Comparing how this specific game determined that a slight improvement had been achieved for at least this game. Considering the time difference, this improvement was not massively significant.

$$\underset{d=6}{\begin{bmatrix} 2 & 8 & 4 & 2 \\ 4 & 16 & 64 & 16 \\ 2048 & 512 & 256 & 32 \\ 4096 & 1024 & 16 & 128 \end{bmatrix}} \quad \underset{d=7}{\begin{bmatrix} 8192 & 512 & 32 & 16 \\ 4096 & 256 & 64 & 8 \\ 1024 & 128 & 16 & 4 \\ 8 & 16 & 8 & 2 \end{bmatrix}} \quad \underset{d=h(x)}{\begin{bmatrix} 8192 & 4096 & 512 & 4 \\ 2048 & 1024 & 64 & 16 \\ 128 & 64 & 32 & 8 \\ 8 & 2 & 4 & 2 \end{bmatrix}}$$

9 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 [16]. This allows for dubious ethical 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 licenses were created (see 9.1).

9.1 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." [16].

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 [17]. 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 [18]; 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 copylefted [16].

9.2 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)

Even though 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 the environment sections. It is not fair on these projects that they don't get the credit they deserve when they enable a task to be done well. 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. If the footage had just been recorded with OBS, the video would have been much longer and not as high quality.

Some of the source code used in the project is 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 without mentioning the open-source software and breaking licenses would not be broken. While legally, this behaviour would be considered acceptable, from an ethical perspective, this would be highly unethical as it would be profiting from another developer's work without even crediting that developer's work.

Currently, no source code has been used under a copyleft license in this project; however, if 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 developer's code, other options do not involve paying for the software, such as compiling from a source or downloading a third-party binary from another source. For a well-researched user, the payment for a binary is a choice rather than the only way of using the software.

While the license does not require it, if this code were ever published, it would be worth putting a link to the 2048 GitHub page [2] in the README file, as it was an inspiration to work on the project. Though 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 Gabriele Cirulli created the concept of 2048.

9.3 Unlicensed Source Code and Copyright

Many publicly available GitHub projects, partially small ones, have never been licensed. In many of these situations, likely, 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 a project like this breaches the owner's 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 the educational exception to copyright covers this project, 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.

10 Self Evaluation

Overall, this project has gone quite well. I have achieved an algorithm that can consistently reach the higher end of what humans can achieve. Some decisions, I think, held this project back, many of which were made very early in the project.

1. Using Java - Java was chosen because it was the most appropriate language I am confident in, not because it was the best language for the job. A faster language such as C++ or Rust would have been more appropriate.
2. The game is tied to closely to the algorithm. Early in the project I decided that the next state of the game would be obtained from the algorithm. This means the algorithm has to keep track of irrelevant data, such as the game score. There are many repeated 'states' in an expectimax tree; however, there are very few situations where the scores are the same. This decision made it awkward and inelegant when I attempted to implement a form of memorisation.
3. Early on, I decided to apply my algorithm to $n \times m$ games. While this was an interesting problem, the best implementation I found [3], managed to store the game in a 64-bit integer.

The operations were done entirely with bit shift and mask operations. The variable-sized game makes optimisations such as that more difficult.

Despite these problems, this implementation has some strengths.

1. The program is very flexible. The MVC design pattern allows new models and views to be made to meet the different needs of the program. When benchmarking performance, the 'benchmark view' can be used, or when optimising a value, the 'optimiser view' can be used.
2. The program can utilise all available resources on a CPU for more resource-intensive games.

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] R. Xiao, "2048-ai," 2014. [Online]. Available: https://github.com/nneonneo/2048-ai

[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] *Executing stream in parallel.* [Online]. Available: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html#executing_streams_in_parallel

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

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

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