Final Year Project Report

Full Unit – Final Report

Development of an Autonomous Agent to Play 2048

Justin Everett

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Ilia Nouretdinov

Department of Computer Science

Royal Holloway, University of London

April 10, 2025

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 12,492

Student Name: Justin Everett

Date of Submission: 10th April 2025

Signature:

Table of Contents

Abstract

Entailed in this report is the progress of the work done in term 1 towards creating an autonomous agent to play the game *2048* [1]. The project has been split into two major component timelines, the first of which occurring in term 1 and involving the creation and implementation of the system which will play the game automatically. The work in term 2 extends this agent to be able to play the game from a website by utilizing computer vision and the *selenium* [2] library.

Chapter 1: **Introduction**

1.1 Project Aims

The goal of this project is to be able to automatically play the sliding tile game *2048* on its official website publication quicker and more accurately than a human. By the end of first term, this had been achieved on a local prototype version of the game but still needed to be extended to be played on the official website. This extension, completed throughout second term, involves utilizing the OpenCV computer vision library [3] and the selenium library to both identify the game board from the website and send input to the website.

1.2 What is *2048*

2048 is a singleplayer tile video game made up of a 4x4 grid of tiles, where the player aims to achieve the highest score they can by combining tiles with up/down/left/right inputs. Tiles combine if they are of the same value and the player's input would cause those tiles to move into the same space, and when 2 tiles combine, they merge into 1 tile whose value is double its constituents. Directional inputs move not only individual tiles, but each tile on the grid to as far as it can move in the input direction, applying as many combinations as possible. Below is an example of what a 'left' directional input during an example game may look like:

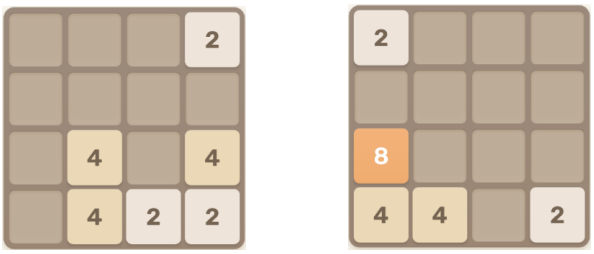


Figure 1. Before and after of a leftward movement input

Additionally, as shown in the bottom right corner, a new tile of value 2 has appeared. This is due to a game mechanic which generates a new tile, of value either 2 or 4 with a .9 probability and .1 probability respectively, at a random empty square on the board after each player move. It is also important to note that combinations are prioritized such that the closest tiles to the edge matching the directional input provided will be the first to combine, and tiles may not combine more than once per turn.

The official goal of the game is to achieve a tile on the game board of value 2048, however the player may continue to play the game with the aim of achieving the highest score or highest value tile they can, up to a maximum possible tile value of 2^{17} or 131,072. Score is calculated by adding to the score the value of any resulting combinations during the turn they are performed. The game ends when no further movement is possible.

1.3 Project Motivation

This project provides an insight into the expanding capabilities of artificial intelligence, specifically the capability of algorithms like expectimax and other similar algorithms to play previously human-dominated games better than even the best humans. This can already be seen to be

happening in several cases, such as IBM's Deep Blue's victory over the former chess world champion Garry Kasparov in 1997 [4] and Google's DeepMind AlphaGo's victory over the Go world champion Lee Se-dol in 2016 [5]. In the end, this project highlights the importance of further development of artificial intelligence, as it can serve as a tool to achieve things beyond the abilities of even the most accomplished humans.

1.4 Why This Project Helps Me

Upon graduating from university, I plan to enter industry in the field of machine learning. This project should be a helpful steppingstone towards that goal, as it builds on my previous interests while helping me develop proficiency with several concepts, tools, and technologies used in the industry. As an example of this, computer vision is an increasingly important aspect of the machine learning field, which the OpenCV library is used extensively for. Having the ability to solve problems and design programs using this library should prove extremely useful in my future career.

Additionally, the completion of a large-scale programming project such as this can be put onto my CV, aiding in my ability to procure employment in-industry, seeing as this project both shows long-term commitment to difficult goals, and is highly relevant to the field I aspire to be a part of.

1.5 Project Timeline

Below is the timeline for this project, divided by term.

Term 1

Week 1:	Study *2048* game, adversarial search, minimax and expectimax algorithms, constraint satisfaction problems, and previous approaches
Week 2:	Create project plan, begin working on creating *2048* prototype
Weeks 3 – 4:	Develop *2048* prototype in its entirety, complete with the ability to output a string representation of the game board at any point
Week 5:	Develop method for handling heuristic evaluation of game board
Weeks 6 – 8:	Implement expectimax algorithm using previously developed heuristic analysis
Week 9:	Connect expectimax algorithm agent to prototype *2048* game
Weeks 10 – 11:	Prepare for interim report and presentation

Term 2

Week 1:	Study computer vision and its use in video games
Week 2:	Develop a test program with OpenCV library to learn how to use it

Weeks 3 – 5:	Create computer vision program for identifying a *2048* game board from a web page
Week 6:	Connect aforementioned computer vision program's output to expectimax agent's input
Week 7:	Use selenium library to relay agent's output to website as a simulated player input
Weeks 8 – 9:	Evaluate performance of complete agent for automatically playing *2048*; compare to similar approaches taken to solve the game by both humans and artificial agents
Weeks 10 – 11:	Prepare for final report and viva

Chapter 2: **Background Theory**

2.1 Adversarial Search

Adversarial search is a search technique used in artificial intelligence which is primarily useful for playing games. It provides a framework for how to autonomously play either single player or multiplayer games by viewing the game as a back-and-forth trade-off between player and adversary turns. The player and adversary are thought to be working against each other, with each of their goals being to maximize their own payoff while minimizing their opponent's payoff.

In these adversarial search problems, after each turn (player or adversary) there are likely to be several possible follow-up scenarios. This can best be expressed through a classic example of an adversarial search problem: chess. After each player's move in a chess game, the opponent often has several different moves they can respond with, and each of these potential responses will have several potential responses, and so on and so forth. Therefore, search problems of this kind can be laid out as tree search problems, with each tree layer alternating between player and adversary turns. The children of any node in the tree correspond to all the next player's possible responses to the move of the parent node. Therefore, the general approach to an adversarial search problem can be shown with the following flow chart:

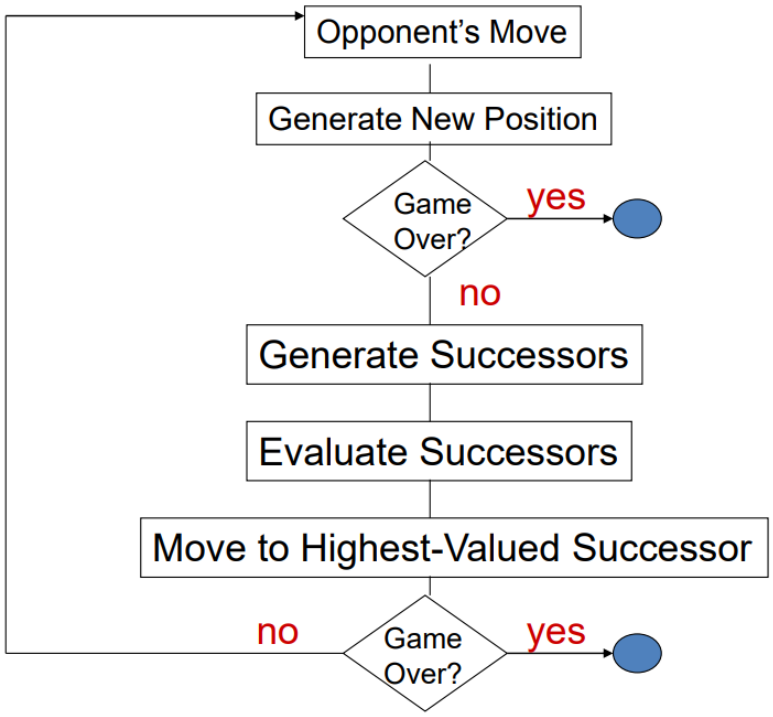


Figure 2. Gameplay loop of an adversarial search problem [6]

2048 can be framed as an adversarial search problem, however where a traditional adversarial search problem has two players alternating turns, *2048* is single player, thus a component of the game must be made to act as the adversary. Conveniently, there is such a mechanic in the game – the random new tile generation. In the process of making a movement in *2048*, all the tiles are moved, then any possible combinations are applied, then the new tile is generated and randomly placed at an empty space. If a hypothetical barrier is applied between the tile movement/combinations and the new tile generation, the game can be thought of as a trade-off between predictable player movement and unpredictable 'opponent' tile placement, satisfying the

conditions of an adversarial search problem. Framing *2048* this way then allows for the application of adversarial search techniques to solve it.

2.2 Minimax Algorithm

The most basic case of an adversarial search problem is that which is entirely deterministic and yields perfect information for all players. An example of this form of adversarial search is the game tic-tac-toe, where each player can know confidently each of their opponent's potential responses to a move. In this deterministic, perfect information scenario, an algorithm called the minimax algorithm can be used to navigate the best moves to make during the game. The underlying premise of the minimax algorithm is that the player's opponent will always play to the player's detriment and will pick the moves which give the least benefit to the player. Under this premise, the opponent's moves can be framed as 'min' nodes, which are tree nodes that always choose their least valued child. Similarly, the player's moves can be framed as 'max' nodes, which always choose their highest valued child. The game is then framed as a tree composed of alternating layers of min and max nodes, shown below:

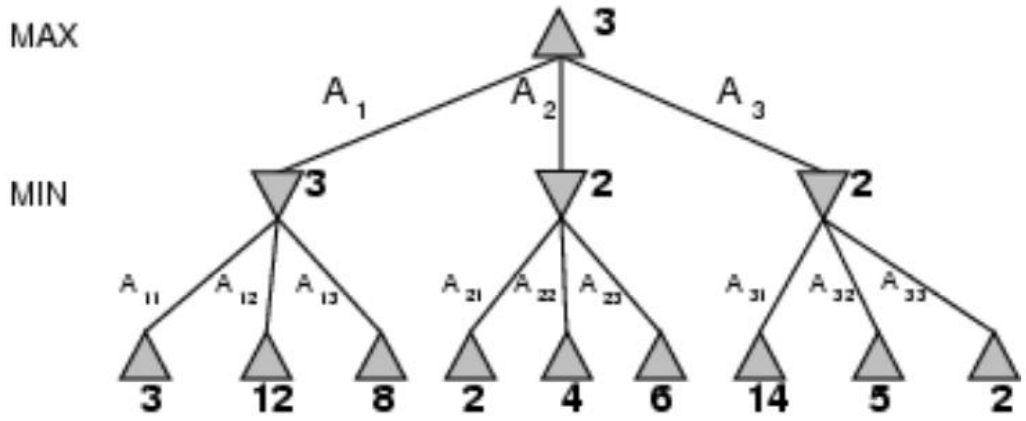


Figure 3. Illustration of the minimax tree structure [6]

As shown in the above example, each of the min nodes minimizes their outcome, then the top max node maximizes from its choice of min nodes. This technique can be applied directly to tic-tac-toe, where the values at each point would represent some score given to the tic-tac-toe board that represents that node. One problem with this, however, is that in a more expansive adversarial search problem, such as chess (which is also deterministic and perfect information), it is often infeasible to search to the end of the game on any branch of the tree. To combat this issue, the depth to which the tree is searched is limited, then the nodes at the maximum search depth are evaluated using heuristic techniques.

The minimax algorithm is extremely adept at handling these deterministic game scenarios but cannot be directly applied to *2048* as the game is not deterministic due to the uncertain nature of new tile generation. The algorithm does, however, lay the foundation for the more advanced expectimax algorithm, which is capable of handling non-determinism, and will be shown later in this report to be extremely well-suited to *2048*.

2.3 Heuristics

Heuristics can be thought of as problem-solving shortcuts or strategies which are used to solve a large or uncertain problem [7]. In scenarios where there are many possible choices, it can be infeasible to check each for optimality. For this reason, heuristics aim to provide a good answer,

but not necessarily the best answer. In the field of artificial intelligence, heuristics are often used for this exact purpose. An example use of heuristics is for the travelling salesman problem, a popular constraint satisfaction problem, where a graph is used to represent locations across a map, connecting location nodes to each other via edges labelled with distance. The goal of the problem is to find a route across the graph which visits each node in the least distance possible, however since the graph is entirely interconnected, the number of different possible routes increases exponentially with the number of graph nodes. Therefore, it quickly becomes infeasible to check each route across the graph, so heuristics are used to find good solutions quickly. An example heuristic for the travelling salesman problem is the nearest neighbours heuristic, which finds a good path across the graph by choosing the nearest unvisited node at each point until the entire graph has been visited.

Since *2048* has several possible moves after each move taken, its complexity increases exponentially with search depth. Because of this, like with the travelling salesman problem, it is infeasible to check every possible subsequent game board, thus heuristics need to be used to aid in determining solutions for the game.

2.4 Recursion

Recursion is a technique where a function is made to call itself. Recursive function calls are held in a stack data structure, e.g. the first call to the function is the last to resolve, and each subsequent call to itself is housed higher on the stack and thus returns quicker. A common example of a recursive process is the process of finding a general term *n* of the Fibonacci sequence, which is F(n) = F(n-1) + F(n-2). While recursion can be useful for some simple arithmetic problems such as this, it excels at creating and navigating search trees.

Most recursive algorithms have a very similar structure consisting of a base case and a recursive case. The base case is the case when the recursive function no longer calls itself, rather returns a value which is then propagated down the stack to feed towards the original function call. In the Fibonacci sequence, this is when *n* is less than or equal to 1, in which case the function return 1 rather than doing a recursive call (since the first two values of the Fibonacci sequence are equal to 1). The recursive case is when the function calls itself, and usually has some diminishing component, such as tree search depth, or in the case of the Fibonacci sequence the decreased values of *n*.

Since recursion is especially useful for tree navigation, it will be essential in the development of the tree-based algorithm which will navigate the game tree for *2048*.

2.5 Human Strategies for *2048*

From *2048*'s simple game rules, a few notable patterns emerge. Firstly, all tiles possible in the game must be of a value equal to a power of 2, since each combination results in the doubling of the original tiles' values. Secondly, as shown by Bhargavi Goel in their mathematical analysis of the game [8], keeping the highest value tile in a corner and working towards combining all tiles toward that corner shows to be an effective strategy component in attaining a higher score. In an article published by Lê Nguyên Hoang [9] also analyzing the math behind *2048*, it is shown that in the most optimal of cases, a 'snake line' pattern throughout the board from one corner to the other where the tiles along the path are successive powers of 2 from 4 up to 2^{16} could be constructed to achieve the highest value tile possible. However, this snake pattern, as pointed out in Bhargavi Goel's analysis, is extremely difficult to achieve as the game *2048* is inherently non-deterministic. Although this snake pattern is difficult to achieve, it propagates the idea of keeping the largest tile in a corner while preventing tiles with values of sequential powers of 2 from being separated by other tiles, and these principles can be used as a guideline for how to play the game, leading to some general human strategies for playing the game.

Below are four of these such strategies, along with the reasoning behind the use of each:

Keep the highest value tile in a corner	Likely the most important strategy, keeping the highest value tile in a corner keeps it in a reliable position and therefore less vulnerable to being displaced by random tile generation. Additionally, keeping the largest tile in a corner makes it easier to combine when an equal-valued tile is eventually created.
Keep next highest value tile immediately adjacent to highest value tile, if possible, same with next highest value, so on and so forth	Keeping the largest tiles sequentially ordered on the game board makes it easier to combine tiles into large values and combines them towards the corner of the largest tile, maintaining the first strategy rule.
Never move in a direction that would take the highest value tile out of the corner	The new tile generated after each move could then possibly generate in the corner where the highest tile is meant to be, making it extremely difficult to put the highest value tile back into the corner, violating the first strategy. This strategy is possible to maintain while playing near-normally if 3 directional inputs are used rather than 4.
As much as possible, prevent two higher value tiles from being separated by a tile of much lower value	If the second strategy has been followed and tiles are lined up sequentially, then allowing for a lower valued tile to intervene between two similarly valued tiles makes it extremely difficult to sequentially combine tiles towards the corner

2.6 Computer Vision

Computer vision is a field of artificial intelligence, responsible for image and video analysis. According to IBM, *"If AI enables computers to think, computer vision enables them to see, observe and understand,"* [10]. The premise of computer vision is the idea that given a machine learning model, if a wide breadth of images are input along with their corresponding labels (e.g. an image of a cat labelled 'cat'), the model can be trained to identify and classify a wide range of visual input.

The field of computer vision has seen a boom in the last 5 or so years, especially with the increasing image recognition capabilities of emerging technologies like OpenAI's ChatGPT [11]. This is resultant from the several applications of computer vision, including image classification, object detection, object tracking (e.g. in a video clip), image segmentation, and context-based image retrieval (i.e. the ability to retrieve an image of a tractor if asked to).

The techniques used to accomplish computer vision fall within the field of machine learning, more specifically the field of deep learning. Deep learning is a subset of machine learning built on the premise of using neural networks to extract patterns from data. Neural networks are designed to mimic the human brain, and thus the way a computer identifies an image is extremely similar to how humans do, except humans use biological components and have had generations of head start to train how our brain should respond to the input our eyes provide, as pointed out in IBM's article on computer vision [10].

Computer vision is relevant to this project because it provides an adaptable way to identify a *2048* game board from an image, allowing the locally built expectimax algorithm in this project to interface smoothly with the website the game is hosted on.

2.6.1 Image Preprocessing

For many computer vision models, raw images often cannot be used as input on their own. Most images being classified, segmented, or detected have too much extraneous noise unrelated to the item(s) being processed, which leads to drastic drops in performance of these models. To combat this issue, a technique called preprocessing is used before the images are attempted to be recognized by the neural network. An example of what preprocessing might entail could be converting a colour image to black and white, which reduces the number of details the neural network needs to focus on and instead provides a much simpler image to identify. There are a few different preprocessing techniques used in this project, which are listed below:

(Gaussian) Blurring	Blurring is exactly as it sounds: the process of making an image blurrier. This is done by changing the colour of each pixel to be the average of its neighboring pixels and can reduce noise in an image dramatically.
Conversion to greyscale	Converting an image can help to reduce noise in the image by simplifying the colour data in an image, forcing the neural network to only have to process entirely gray images. This makes features that are colour-independent more prominent.
Conversion to black and white (binary)	Conversion to black and white, or *binarization*, is a more extreme form of the greyscale preprocessing method, additionally reducing noise and simplifying the image by ignoring the brightness of pixels and instead only using solid black or solid white. This also makes colour-independent features more prominent.
Edge detection	Edge detection is a technique that can be used on binarized images, where objects in an image only have their outline preserved. This can be useful for identifying certain images, however the tiles no longer being filled in can lead to a reduction in performance if applied incorrectly.

2.6.2 Optical Character Recognition (OCR)

Optical character recognition, as detailed by AWS [12] is the process of converting text in an image to a computer-readable format (i.e. a pdf document). This is done via computer vision neural networks like described in the above sections, trained to classify text characters from an image. Some technologies used for OCR include neural networks trained on the popular MNIST dataset [13] (a dataset of binarized handwritten digits), and Tesseract, an OCR engine designed to identify full lines of text from images. Tesseract is a very convenient tool as it is pre-trained to be able to "… recognize more than 100 languages 'out of the box'," [14], eliminating the lengthy training process involved in designing an OCR neural network.

Tesseract is built on a type of neural network called a long-short-term-memory (LSTM) network [14], which is slightly beyond the scope of this paper but effectively provides a neural network to retain memories from previous components of a sequential piece of data. It does not handle preprocessing, so to achieve a correct output preprocessing needs to be applied before use. Additionally, a wrapper for the Tesseract engine exists in python, allowing for easy integration into python-based programs.

The Tesseract OCR engine (and its python wrapper, *pytesseract* [15]) is important to this project as it provides a reliable way to identify the digits held within the tiles on a *2048* game board, which will be seen later to be crucial to the design of this project.

2.7 Selenium

Created originally by Jason Huggins in 2004, Selenium is a suite of open-source browser automation tools, which has been developed and updated actively by its large community of users since its creation. Its most used application is its WebDriver, which *"… drives a browser natively, as a user would, either locally or on a remote machine using the Selenium server,"* [2] allowing for easy automation of user interaction with web pages, such as button clicks, sending inputs, web page navigation, and more. WebDriver works by sending JSON requests to a browser driver, then receiving HTTP responses and sending those back to the user. It is compatible with several widely used browsers like Chrome, Firefox, Edge, etc., and it supports several programming languages including Python.

Similar technologies do exist which achieve very similar goals, such as Puppeteer [16] and Playwright [17], however these are not fit as Selenium for this project. Puppeteer is unfit as it only supports the JavaScript language and the use of Node.js, which would needlessly complicate what is meant to be a simple component of this project, as the webpage interfacing requirements of this project are quite lightweight. Playwright is much more modern technology than Selenium, having been released in 2020 by Microsoft, and like Selenium it supports most commonly used browsers and provides support for Python. Its architecture differs from Selenium in that instead of JSON and HTTP it uses a WebSocket connection which stays open for the duration it is used [18], providing slightly faster execution speed. Although Playwright sounds promising, Selenium's WebSocket proved easier to learn, has been around for longer thus providing higher reputability, and generally fit the needs of this project more closely. Additionally, since the component of this project where Selenium is used is very lightweight, the small increase in execution speed provided by Playwright would result in a negligible increase in speed of the agent as a whole. For these reasons, Selenium was chosen for use in this Project.

Selenium is crucial to the design of this project, as without it (or some similar technology like Playwright) playing *2048* automatically via a website becomes extremely complex.

Chapter 3: **Software Engineering**

Term 1

3.1 *2048* Prototype

The starting point for this project was a 'prototype' local version of the game *2048*. The goal of creating this prototype was to be able to create and test the expectimax algorithm on the game without first needing to develop a method of interfacing with the website that hosts the game. The basis of this implementation was the use of the python library *Pygame* [19], which is widely used for developing simple games with python.

The first step of development was the creation of the game window, event handlers, and main gameplay loop, all standard procedures with Pygame. After the skeleton of the game had been set up, the next addition was two classes: Game_Board and Game_Tile. These are both massively impactful on the rest of the project's code as they contain all the logic used to play the game and are used by the expectimax algorithm to represent game boards. The Game_Tile class is a small class whose instances represent a singular tile within a *2048* game and contain a value and associated tile colour. The Game_Board class utilizes 16 Game_Tile objects to construct a 4x4 game board grid and holds these tiles in a 2-dimensional array representing the current condition of the game.

3.1.1 Tile Movement

Having developed the capability for the game board and tiles to be displayed, the next step towards emulating the original game was implementing the directional tile movement. This involved adding 4 movement methods to the Game_Board class, each of which works by going row by row or column by column through the 4x4 matrix representation of the board. The columns are obtained by transposing the matrix then iterating through the transposed rows.

These rows and columns are processed by extracting their nonzero tiles and replacing the now missing tiles in the resulting row/column with zero valued tiles, e.g. a row [2,0,4,0] becomes [2,4], then zeroes get filled in either before the 2 or after the 4 depending on if the movement is left or right. The one hitch in implementing these movement methods is the necessity to check whether a movement is successful, meaning that the resulting game board after a movement must be different to its predecessor, otherwise the movement has done nothing and should not be counted.

3.1.2 Random Tile Generation

Each movement made by the player must then be immediately followed by a new tile being randomly added to the game board, as per the original *2048* game. This mechanism was implemented into the prototype by generating a new tile instance whose value is randomly decided to be either 2 or 4, at a 90% or 10% probability respectively. This tile is then attempted to be placed at a position on the game board corresponding to a randomly generated number between 0 and 15. If the position is already occupied by a non-zero tile, then the position number is incremented by one, and is checked for availability again. This continues until either the new tile has been placed onto the board, or until the loop executes 16 times, at which point it can be deduced that all the board tiles are nonzero, and a new tile cannot be placed.

3.1.3 Tile Combination

With movement and random tile generation completed, the last major step for the game to be playable is the tile combination feature, which dictates the consolidation of two colliding equal valued tiles into one tile of double the shared value. This was achieved via a method which takes in

the direction of movement as an argument, then splitting the game board into rows or columns depending on the direction provided and looping through each of these rows/columns and combining any adjacent tiles into a new tile of doubled value. An excerpt of the source code for this method is shown below:

```
result_list = [Game_Tile(0)] * len(tiles)
if direction in ('right', 'down'):
    i = len(result_list) - 1
    j = len(tiles) - 1
    while i >= 0:
        if i > 0 and tiles[i].value == tiles[i - 1].value:
            new_value = tiles[i].value * 2
            result_list[j] = Game_Tile(new_value)
            self.score += new_value
            i -= 1
        else:
            result_list[j] = tiles[i]
        i -= 1
        j -= 1
```

An interesting feature of this method is that since the columns are obtained by transposing the 4x4 game board matrix then looping through them as if they were rows, the code for the combinations in the left direction is identical to the code for combinations in the up direction, and the same goes for right/down, as shown in the conditional on the 3rd line of the excerpt above.

3.1.4 Supplemental Methods

The finishing touches to the prototype version of *2048* include a method which checks if the game is over by performing a movement in each direction and checking if the board resulting from those movements is the same as the current board, and if so then no movements can be performed, thus the player loses, and the game is over. Methods for checking equality of two game boards and copying a game board were also created, primarily for the game-over/movement-success checks and for logic checks in the development of later heuristics and expectimax algorithms. Lastly, the Game_Board class includes two methods for converting the Game_Board object into a 1-dimensional array or a 2-dimensional array, which are used later for the heuristics and expectimax implementations.

3.2 Heuristics for *2048*

Having developed an interactable local version of *2048*, the first step towards automatically deciding the best move to make is the development of the heuristics to be used in the heuristic analysis step of the expectimax algorithm. These heuristics will be used to assign a numerical value to a game board via a weighted sum of features, allowing different boards to be rigidly compared to each other.

The weighted sum of features that comprise the general heuristic analysis are each based on the ideal strategies of *2048*, and are inspired in part by Kohler, Migler, and Khosmood's paper *Composition of Basic Heuristics for the Game 2048* [20], which suggests five heuristic strategies for *2048*: greedy, empty, uniformity, monotonicity, and random. For the heuristics of this project, six strategies are used together in a weighted sum on the same game board to produce the evaluation for that board, which are the greedy, empty, uniformity, and monotonicity strategies, along with two boolean strategies which return the maximum reward if the largest tile on the game board is in a corner, and if the sequence of the (up to) four largest tiles on the board are in the same row or column, which will be referred to as the corner and order strategies, respectively. Therefore, the heuristics used to evaluate a game board in this project include the greedy, empty, uniformity, monotonicity, corner, and order strategies, and their implementations are described below.

3.2.1 Greedy Heuristic Strategy

Like most greedy strategies, the greedy heuristic implemented in this project is simply the score of the current game board. Before being handed off to the evaluator, however, (which sums together several features), this score must be normalized to a value. It is important to do this because as tiles combine and add to the score, their values increase in powers of 2, resulting in a non-linear increase in the score which must be then scaled down to a linear value via a logarithm. This gives a simple immediate determination of how 'good' this board is. Additionally, since score is only increased when tiles are combined, an emergent effect of this simple greedy heuristic is that evaluation is rewarded for tile combinations and awarded more-so from large tile combinations.

3.2.2 Empty Heuristic Strategy

The empty heuristic strategy is another quite simple strategy, which increasingly rewards for how many empty tiles are present on the current game board. This is advantageous to the *2048* agent because it promotes having more empty space available than not, thus keeping the agent further from the game ending. As a side effect of promoting empty space on the board, this strategy also promotes the act of combining tiles, as the act of combining tiles reduces the number of non-empty spaces on the board.

3.2.3 Uniformity Heuristic Strategy

Uniformity is a strategy which rewards for having many equal-valued tiles. The implementation of this strategy has been done by simply returning the number of occurrences of the most frequently occurring tile on the game board; doing so rewards the agent for setting up potential combinations. This is the case because with more equal-valued tiles on the board, there is more opportunity for each of these tiles to collide with each other and combine. With more opportunity for combination, the agent is more likely to score higher on both the empty and greedy heuristics, hence the benefit of the uniformity heuristic.

3.2.4 Monotonicity Heuristic Strategy

The monotonicity heuristic is a more complex strategy than the previous three, the premise of which is to check whether the current game board is either completely non-increasing or completely non-decreasing, returning a normalized value (between 0 and 10) representing the number of tiles on the game board that do not conform to this monotonicity property.

Figure 4. Example of a highly monotonous game board

Figure 4 displays an example of a highly monotonous game board, which can be verified by looking at each row and column, which are all non-increasing. The implementation of this heuristic in this project's source code works by keeping a rolling tally of the number of non-conforming tiles for both the non-increasing and non-decreasing monotonicity properties, then iterating through each row and column of the game board and checking whether the difference between each pair of tiles in the row/column is positive or negative and using this information to determine whether the tile being evaluated conforms. Importantly, since empty tiles are represented as tile objects with value 0, when a 0 is encountered it must be ignored otherwise empty tiles may falsely contribute to

the number of non-conforming tiles. Below is an excerpt from the source code for the monotonicity heuristic, showing how a non-decreasing row check is performed:

```
for row in self.board_array_2d:
    previous_value = row[0]
    for value in row:
        if value > 0:
            nonzero_tiles += 1
        if value != 0:
            if value < previous_value:
                non_conforming_increasing += 1
                previous_value = value
```

If a column is being evaluated rather than a row, the process is identical par from the grid variable being evaluated, which is a transposed version of the 4x4 game board grid, which results in the columns taking the place of the rows in the grid, allowing the iteration process to remain the same. The process of transposing the grid is shown below, with *.T* being a NumPy method that transposes a matrix:

```
columns_transposed = self.board_array_2d.T
```

Utilizing the monotonicity heuristic is primarily for rewarding the agent for setting up future potential combinations, as a highly monotonous game board has the largest tiles sequentially lined up, mimicking the 'snake line' strategy to a degree, therefore setting up the largest tile on the board for combination. Monotonicity also promotes higher scores on the greedy and empty heuristics, as having the largest (and least likely to be combined) tiles pushed into the corner leaves more continuous space on the game board for smaller tiles to move and combine with each other.

3.2.5 Corner Heuristic Strategy

An expressly boolean heuristic strategy, the corner strategy rewards the agent heavily if and only if the largest tile on the game board is in one of the corners. Although this strategy is already encouraged via the monotonicity heuristic, having the largest tile remain in the corner is such an important component of effectively playing *2048* that the agent should almost never move the largest tile out of the corner under any circumstances, hence the necessity for the corner heuristic strategy which heavily advantages such game boards.

3.2.6 Order Heuristic Strategy

The order heuristic strategy is the last of the heuristics used to evaluate a game board's strength and is another boolean strategy. This heuristic rewards the agent if and only if the largest (up to) four tiles on the board are in the same row/column, therefore promoting further the 'snake line' strategy. This has been implemented by sorting all the tile values on the game board, then using the number of non-zero tiles to extract the top (up to) four tiles. Each row and column are then iterated through to check if these values occur in the same row/column, and if so, the agent is rewarded heavily.

3.3 Expectimax Algorithm

Having implemented the local *2048* prototype and heuristics for evaluating a game board, the last component of the first term timeline was to implement the expectimax algorithm. This algorithm construes the bulk behind automatically finding the best moves to make from a given game board and is an extension of the minimax algorithm. Where the minimax algorithm assumes determinism and optimal play from the adversary the expectimax algorithm does not, making it well-adapted for *2048*'s non-determinism and random 'adversary play'. A key distinction, however, between traditional adversarial search problems and the context of *2048* is that *2048* is singleplayer and does not have a strict 'adversary'. Instead, the random generation of new tiles functions as the adversarial element, introducing uncertainty.

To address this uncertainty, the expectimax algorithm incorporates probabilistic outcomes into its decision-making process, allowing not only for the evaluation of immediate outcomes, but also the likelihoods of future board states occurring. The algorithm achieves this by recursively generating all possible game boards that carry from the current game board, alternating between two key modes: player moves and random tile generation. Player moves are represented as max nodes (as in the minimax algorithm) and importantly do not construe a full movement in the game, rather simply a movement before the new tile has been generated and placed. The random tile generation is represented as expectimax chance nodes, which determine the average expected evaluation of all the node's children. A basic illustration of the tree structure of the algorithm is depicted below:

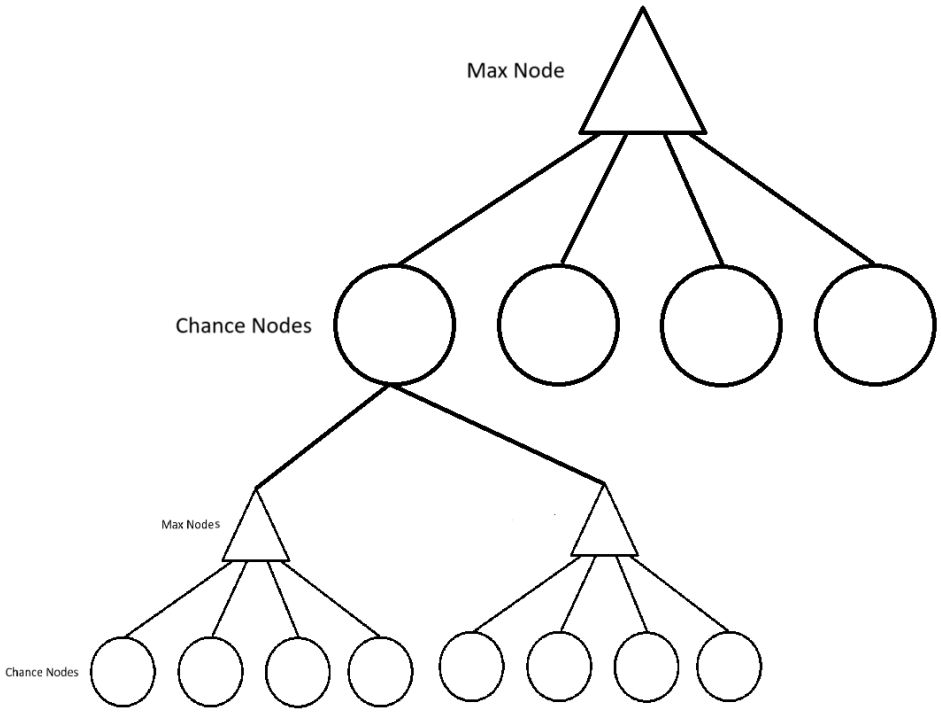


Figure 5. Illustration of the expectimax algorithm tree structure

3.3.1 General Structure of Algorithm

As the expectimax algorithm is a recursive tree traversal algorithm, its structure follows that of most recursive algorithms, having a base case and recursive cases.

The base case for this algorithm is any scenario which a terminal state has been reached, i.e. when a game board needs to be evaluated using the previously defined heuristics. This is the case under two circumstances: if the maximum search depth has been reached, and if the game is lost. Additionally, a second base case is added if the current board is null, which is included to handle the case when an attempted move is unsuccessful (e.g. a move to the right doesn't change the board, meaning no rightward move is possible and thus should not have been attempted); this case always evaluates to 0. The source code for these base cases is as follows:

```
if board is None:
    return None, 0
self.evaluator.set_board(board)
if depth == 0 or board.is_game_over():
    return None, self.evaluator.evaluate()
```

This algorithm has two recursive cases; one case for player moves and the other for tile generation. They are differentiated via a boolean which communicates the current turn status through the recursive function calls, however this information cannot be directly sent through to recursive calls without including the information in the recursive function call itself. Therefore, as shown below,

the turn status boolean is included in the parameters of the expectimax function and is by default true since the algorithm will always start with a player turn.

```
def expectimax(self, board, depth, is_player_turn=True): (...)
```

Also shown in the function declaration for this expectimax algorithm implementation is both the current game board and the current search depth. The board is necessary to include as it will need to be evaluated if a terminal state has been reached, and including the search depth is vital because tracking the current search depth and decrementing it with every subsequent recursive call is what prevents the method from infinitely repeating.

3.3.2 Player Move Max Nodes

As the first recursive case in the algorithm, the player move max nodes are handled by generating new game boards from applying each potential directional movement to the current game board, then omitting the new tile generation process. For each of these new game boards, the movement to achieve them is stored, then a recursive call is made to evaluate the new board. The value returned by the recursive call is the evaluation result on that new board, which is then checked against the current best evaluation stored. If its evaluation eclipses the current best, then that move is saved as the current best move. Once each of the potential moves have been checked, the best move and its evaluation are returned. Below is the source code which achieves this:

```
max_value = -float('inf')
moves = ['left', 'right', 'up', 'down']
best_move = None

for move in moves:
    new_board, success = self.get_next_board(board, move)
    if success:
        _, value = self.expectimax(new_board, depth - 1,
                                   is_player_turn=False)
        if value > max_value:
            max_value = value
            best_move = move
return best_move, max_value
```

3.3.3 Tile Generation Chance Nodes

The last step in implementing a functioning expectimax algorithm is handling of the random tile generation chance nodes. These nodes function similarly in principle to the player move nodes in that they recursively check each possible new board and utilize its expectimax evaluation, however where the player move max nodes simply take the best child, the chance nodes have no determination of whether a certain child will be chosen, rather a combination of the expected values of its children and the probabilities that they occur. This is due to the inherently non-deterministic nature of new tile placement; thus, the chance nodes return an expected average result of their children rather than a firm evaluation. This expected value is achieved via the formula for the expected value of a discrete random variable [21], which is:

$$E(X) = \Sigma\, x \cdot P(X = x).$$

In layman's terms, this formula states that the expected value of a scenario X (such as the resulting boards from a random tile placement) is the sum of all possible outcomes of X, each weighted by its probability of occurrence. For 2048, this means the value of a chance node is calculated as the sum of the evaluation results of its child boards multiplied by their respective probabilities. Shown below is how this has been implemented in code:

```
empty_cells = [(i, j) for i in range(4) for j in range(4) if
                                          board_array_2d[i][j] == 0]
```

```
if not empty_cells:
    return None, self.evaluator.evaluate()
expected_value = 0
for cell in empty_cells:
    for tile_value, probability in [(2, 0.9), (4, 0.1)]:
        new_board = board.copy()
        new_board.set_tile(cell[0], cell[1], tile_value)
        _, value = self.expectimax(new_board, depth - 1,
                                                    is_player_turn=True)
        expected_value += probability * value / len(empty_cells)
return None, expected_value
```

The evaluation result comes from the result of the expectimax algorithm recursive call on the child board being evaluated as with the player move max nodes. The probability of each board is derived by considering all possible scenarios where a new tile can be generated, which include an equal probability of a tile occurring at any given empty space, multiplied by the probability of the tile being either a two or a four, which is 0.9 and 0.1 respectively. Therefore, the line which generates the value of the chance nodes is shown above to sum up each of these scenarios, adhering to the expected value formula.

In combination with the player move max nodes, the random tile chance nodes complete the agent's ability to automatically determine a move to make from any *2048* game board, rendering the term 1 timeline complete.

Term 2

3.4 OpenCV Computer Vision

Metaphorically, the aim of term 2 is to give the brain of my program a pair of eyes. By this, I mean that term 1 culminates in the completion of a *2048* expectimax solver which can play the game on its own, however it is missing the sight that allows it to interact with the outside world; the solver-website computer vision interface. This section will entail the development and functionality of this component.

3.4.1 Prototype Program

The first step towards developing the computer vision component of this project is becoming adept with OpenCV. This was to be done via a prototype program whose aim was simply to detect the digits on a screenshot of a browser window. To achieve this goal, OpenCV's documentation, along with a restack article detailing how to do simple digit recognition [22] were used as a starting point. Test images were manually imported and displayed using OpenCV, then pytesseract was implemented to handle the digit recognition itself. The first run of this program on a screenshot of a browser window produced surprisingly good results – some digits on the webpage were being picked up and the image was being displayed properly, however not all the digits were being picked up, and importantly none of the digits within the *2048* game board were identified at all.

This was not ideal, since being able to detect tile values via OCR is necessary to reconstruct the board. To attempt a fix for this, the image was pre-processed to make it easier for tesseract to identify digits. A gaussian blur is first applied to the image to smooth out any edges and reduce noise. Then, the resulting image is made greyscale and subsequently converted into a binary black and white image using a technique called Otsu's thresholding. Otsu's thresholding works by taking a threshold value between 0 and 256 in a greyscale image, then going through each pixel in the image and forcing it to 0 if the pixel's value is lower than the threshold, 256 otherwise. The resulting black and white image only keeps the whitest pixels from the original image, resulting in a clear and simple image which can be fed to the tesseract OCR; however this technique is prone to information losses, which became evident when even this preprocessing didn't detect any of the values in the input image's game board.

To determine what exactly was going wrong, I swapped my input image from a screenshot of a browser window to a cropped-out *2048* game board. Despite only having the game board to look at, even after pre-processing this new image had issues. The tile values were showing up, however some were missing. Specifically, tiles of value 16, 32, or 64 were nowhere to be seen. This is due to the colour scheme of the tiles, each of which has a foreground and background colour. Most tiles in the game are displayed as a dark-coloured number on a light-coloured background, however for the 16, 32, 64, and 4096 tiles (as well as all values larger than 4096) the inverse is true, with a light-coloured number on a dark-coloured background. As a result of this, when Otsu's thresholding is applied to the entire game board image, the dark-on-light tiles are binarized properly and readably retain their digits whereas the light-on-dark tiles' information blends in with the background and is unreadable.

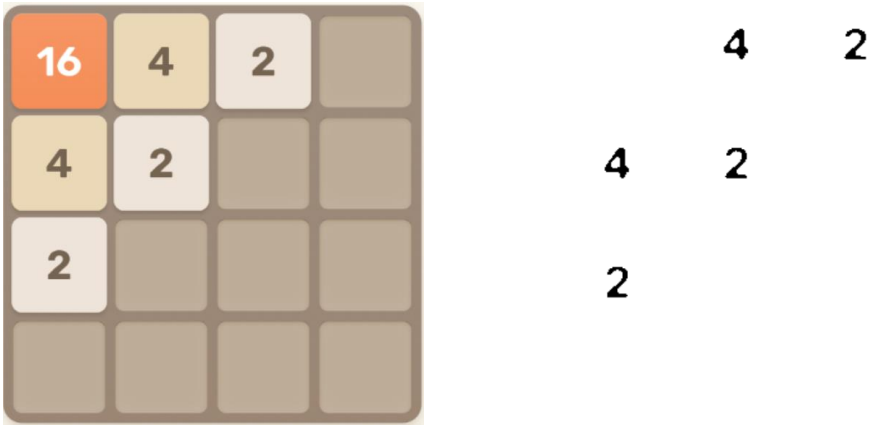


Figure 6. Before and after of 'lossy' game board image preprocessing

As seen in *Figure 6* when preprocessing is applied to the game board, the 2 and 4 tile values are all made more distinct, however the 16 tile disappears completely.

After some testing and tinkering with my preprocessing, I managed to determine that if this preprocessing technique is applied to a single tile rather than the entire image, the information is not lost, and the tile shows up. An example of this is shown below, with both a 16-valued tile and a 4-valued tile being pre-processed individually.

Figure 7. 4 and 16 valued tiles before and after individual preprocessing

As can be seen in *Figure 7*, when pre-processed individually the tiles have no issues with missing values. Additionally, when fed to the pytesseract OCR, the digits are detected nearly-perfectly*, which concluded the prototyping stage.

*As will be seen later, there still exists a small issue with applying OCR to 32 and 512 tiles

3.4.2 Game Board Isolation

Having established a strategy to be able to identify the value of an isolated tile image, the next step was to determine how to isolate individual tiles so they can be evaluated. To achieve this, a technique called contouring was the most promising idea. Contouring is the process of finding the bounding box of a component of an image and is frequently used in deep learning image classification problems. Conveniently, OpenCV has a built-in method *findContours()* which can find all the contours in an image. This method works best with binary images, so it is recommended to apply either thresholding or a technique called canny edge detection first. Canny edge detection is like thresholding in that it produces a black and white binary image, however where the thresholding techniques leave a shape filled in (for example the solid black 4 in *Figure 7*) edge detection only leaves the outline of the shape.

With contouring having been established as the methodology for isolating a game tile, two slightly different strategies emerge as options for implementation:

- Find the contours on an entire binarized game board image, then slice the image by each contour to isolate each tile

- Isolate the entire game board from its original screenshot image via contouring, then manually segment the isolated game board into individual tiles

Clearly the former of these strategies would be simpler, as it avoids the game board isolation step, however there were several issues when attempting to implement this strategy. Firstly, when the image is pre-processed, as seen previously the background of the 16, 32, and 64 valued tiles disappears. As a result of this, contouring these tiles via this method is completely impossible, despite the rest of the tiles being contoured perfectly.

The latter of these strategies, isolating the game board then applying manual segmentation, proved to be both the final option before going back to the drawing board, as well as the most robust option. Importantly, if attempting to isolate the game board from a screenshot, the preprocessing does not cause loss of vital information, as only the outline of the game board is required. Therefore, edge detection works better than thresholding if the goal is to contour the game board, so this preprocessing is applied. However, an issue does arise when finding the contours of the pre-processed image: the game board is contoured, but so is every single other prominent item in the screenshot.

To identify which contour corresponds with the outline of the game board, we can use known properties of the game board to filter out the unnecessary contours. Specifically, we know that the game board should be square (though the length and height may differ by a few pixels due to the contouring process), and the game board contour should be quite large (it should be the second largest contour, only smaller than the contour around the entire window). Using this knowledge, the game board can be isolated by excluding any contours whose width and height differ by too much, and by excluding any contours whose width is less than some arbitrarily large pixel value (such that the tiles are removed, and the board remains). After some testing with different window sizes, the width and height should not differ by more than 100 pixels, and the width/height should be larger than 300 pixels. The code implementation of this is as follows:

```
for cnt in contours:
            #Calculate the bounding box of the contour
            x, y, w, h = cv.boundingRect(cnt)
              #Limit displayed contours to only squares above a certain
size to isolate only the game board
            if w - h < 100 and w > 300 and x > 10:
                #Save rectangle position and dimensions
                self.game_board_rectangle = (x, y, w, h)
```

With the game board successfully isolated, the next step is separation into individual tiles.

3.4.3 Tile Segmentation and Processing

By slicing the original screenshot image using the coordinates of the game board, a new image is produced containing only the game board. Using this image and the knowledge that every *2048* game board is comprised of 16 tiles in a 4 by 4 layout, we can manually divide the width and height by 4, then create a list of tiles by slicing the game board image, as shown below:

```
#Divide board into 4x4 grid of tiles
board_px_width = len(self.contoured_board[0])
board_px_height = len(self.contoured_board[1])
tile_width = board_px_width // 4
tile_height = board_px_height // 4
tiles = []
#Add each tile to the list of tiles
for i in range(4):
      for j in range(4):
            start_y, start_x = tile_height * i, tile_width * j
            tiles.append(self.contoured_board[start_y:start_y+tile_height,
                        start_x:start_x+tile_width])
```

Incidentally, since it is done sequentially, segmenting the tiles in this way preserves the location of each tile, allowing for easier reconstruction of the board after identifying the digits in each tile.

Having isolated each tile, the last step towards being able to fully reconstruct the game board is applying the appropriate preprocessing to the tile before applying OCR. As with the preprocessing of the screenshot image, we need to first apply both a gaussian blur and to make the tile image greyscale. Next, however, instead of applying canny edge detection like was done to isolate the game board, the tiles need to use thresholding so that the digits are filled in, otherwise the OCR performance drops significantly.

Each tile's location and value can now be determined; however, the accuracy of this process is still quite low. This is due to the portions of the game board borders which exist in each tile image creating noise which disrupts the accurate prediction made by the OCR. An example of this can be seen below, with a manually segmented tile before it is preprocessed:

Figure 8. A manually segmented tile before pre-processing

The fix to this issue is simply to crop each tile image such that the digits remain, and the borders don't. How much to crop out was determined by applying increasing amounts of cropping in both the *x* and *y* axes until the values no longer get detected by the OCR. This solution works very well for all tile values; however, it sometimes causes misinterpretation of empty tiles as having value 7. The reason for this is that when the vertical and horizontal portions of the tile border are cropped out, the corner can remain, leaving a single feature in the tile image which the OCR interprets as a 7. An example of this is shown below:

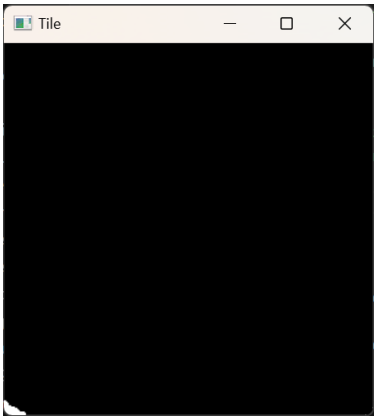


Figure 9. Empty tile after preprocessing which is misidentified as containing a 7

This was fixed quickly by cropping much more in the *y* axis than the *x* axis, as every multi-digit tile value is taller than it is wide.

The other issue present in this process is the OCR sometimes misidentifying 32 and 512 tiles; however, they are both misidentified in a very consistent way, such that 32 tiles can be misidentified as either 52 or 352, and the 512 tile can be misidentified as 912. To solve this, I modified the Game_Tile class to simply map 52 and 352 to 32, and 912 to 512, as shown below:

```
problem_values = {52:32, 352:32, 912:512}
self.value = value if value not in problem_values.keys() else problem_values[value]
```

Since 52, 352, and 912 are not valid tile values, this solution does not take away any valid tile-value mappings, and this procedure works flawlessly.

Having made these changes, a list accurately containing the values of each tile (including empty tiles) can now be constructed from a screenshot of a *2048* webpage.

3.4.4 OCR-enabled Board Reconstruction | Optimizations

With a reliable method of identifying the value of each tile in a game board from a screenshot, all that remains to complete the computer vision component of this project is reconstructing this list of tile values into a *Game_Board* object instance, which can then be fed to the expectimax algorithm developed in term 1.

Conveniently, since my implementation of the *Game_Board* class has the option of taking a list of tile values as a parameter and instantiating a *Game_Board* object with the same tile values, the list of tile values found with the OCR is already sufficient to make a *Game_Board*. However, when testing the process of applying the entire board recognition process to a *2048* webpage screenshot, it was immediately obvious that the process of applying preprocessing and OCR to each individual tile was far too slow. The process took around 3 seconds to produce a *Game_Board* from a screenshot, and at a speed of 3 seconds per move my goal of making the autonomous agent's speed comparable to that of a human would not be satisfied.

To combat this issue, I first tried reducing the amount of 'redundant' preprocessing steps being applied, by applying the gaussian blur and grayscale to the original screenshot image, then simply reusing this grayscale image when applying manual tile segmentation so that each individual tile needn't have a gaussian blur and grayscale applied individually. This idea seems promising in principle, however in practice it does not work as intended, with the OCR accuracy plummeting in response to the modified preprocessing. In fact, any method of preprocessing apart from the gaussian blur and grayscale applied to the board, then individually to each tile, followed by canny

edge detection for the board and Otsu's thresholding for the tiles, proved to be unreliable and inaccurate. Therefore, computation time could not be saved by adjusting the preprocessing steps.

If the preprocessing could not be changed, then the next best option would be to reduce the number of tiles going through the OCR process. Since the result of a directional movement before a new tile has been generated is deterministic, if the previous board is known when undergoing board reconstruction, the location of the empty tiles (and thus the possible spaces for new tile generation) after a move can be determined. These empty tiles are actually the only tiles which need to be checked after each move, as the value in every non-empty tile after a movement can be calculated from the previous board. To help with reducing the number of tiles being preprocessed, a simple method is added to the *Game_Board* class which returns a list containing the indices of each empty tile on the board:

```
empty_indices = []
for index, tile_value in enumerate(self.get_board_array_1d()):
      if tile_value == 0:
            empty_indices.append(index)
return empty_indices
```

Given that there already exists a method for creating the game board resultant from a movement before a tile is generated (*move_left(), move_right(), ...)*, the indices of the tiles which need to be checked for the newly generated tile after a movement can be found by sequentially applying a movement method (i.e. *move_left()*) then the newly created *find_empty_tile_indices()*. This returns a list of all the possible locations where a new tile could generate after a given directional movement, and thus once the list of individual tiles has been created from the screenshot provided to my computer vision component, preprocessing is applied by empty tile index rather than to all tiles. Once a tile is found within the empty tile indices, we know with certainty that this is the newly generated tile, so this tile can simply be added to the board state after the movement (i.e. *move_left()*), providing an accurately reconstructed game board with much less time spent preprocessing. This optimization resulted in movements made by the agent taking ~0.5s rather than ~3s.

Having achieved reasonably efficient board reconstruction, the computer vision component of this autonomous agent is complete.

3.5 Selenium Webpage Interfacing

Although the agent can now generate recommended movements from a screenshot automatically, the component which interfaces with the website itself still needs to be developed. Up until now, screenshots were taken manually, and inputs were provided back to the website manually, so the last component of this project is the development of a simple selenium-based interface between the agent and the website.

3.5.1 Clearing Irrelevant Popups

Selenium's webdriver module makes it incredibly simple to open a browser window to a given website url, however a few things need to be done before screenshots can be taken and moves can be made. Upon opening the *2048* official website for the first time on a browser, the user is prompted with two popups, which both can only be cleared by clicking their respective close buttons. The first of these popups asks the user to accept cookies, shown below, can be cleared by pressing the "Allow Necessary Cookies & Continue" button.

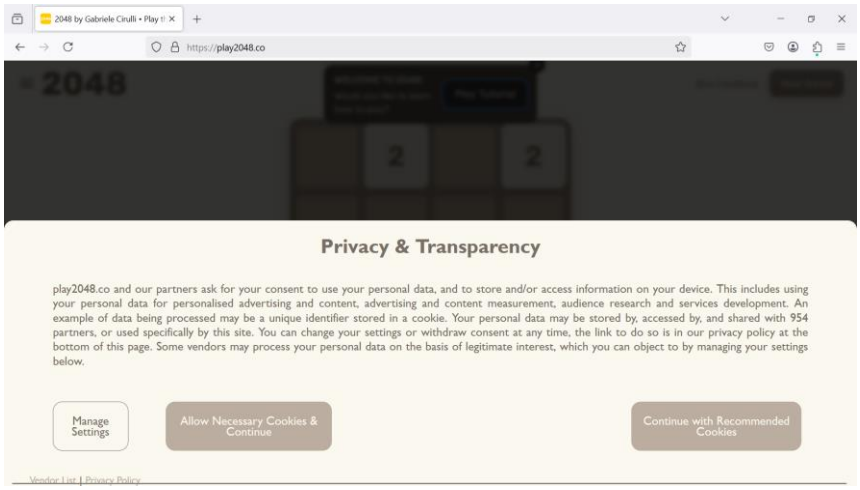


Figure 10. 2048 website cookies popup [source]

To clear this popup, the selenium webdriver is made to wait until a clickable element containing the text "Allow Necessary Cookies & Continue" is found, which is then clicked. A similar process can be applied to the second popup, which prompts the user to complete a tutorial before playing the game. The tutorial popup, like the cookie popup, obstructs part of the game board, thus making it necessary to clear before computer vision can be applied. The code for clearing both popups is shown below:

```
wait = WebDriverWait(driver, 5)
cookie_button = wait.until(EC.element_to_be_clickable((By.XPATH,
"//button[contains(text(),'Allow Necessary Cookies & Continue')]")))

cookie_button.click()
close_button = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".bg-near-black.rounded-full")))

close_button.click()
```

Notably, the tutorial popup close button is found in a slightly different way than the cookie button; it does not contain any searchable text, rather an image of an "X" within a circle, so the button element must be found via the CSS style classes applied to it. To distinctly identify this button from the others on the webpage, two CSS classes are required: *bg-near-black* and *rounded-full*.

Both the cookie button and the tutorial close button are clicked, providing unobstructed access to the game.

3.5.2 Taking Screenshots and Sending Keyboard Input

The last step towards creating a fully autonomous *2048* agent is to construct the methodology for taking screenshots and sending keyboard inputs. Both are remarkably simple.

Taking screenshots from the website is so simple it can be done in one line of code. Selenium's webdriver package provides a method *save_screenshot()*, which takes a filepath as a parameter and saves a screenshot of the web browser to that filepath, overriding the previously stored screenshot. This is a marvelously convenient coincidence, as only 1 screenshot is ever required at a time. The code for screenshotting the webpage is as follows:

```
driver.save_screenshot('src//img//selenium_img//screenshot.jpg')
```

The file type saved is .jpg, as it provides more compression than .png image files, thus causing the preprocessing steps in the computer vision component to run more quickly.

Sending keyboard inputs is ever so slightly more complex than taking screenshots, being done in 3 lines rather than 1. Firstly, the component of the webpage containing the game board needs to be identified so input can be sent to it, which is found similarly to the popup buttons. Next, a dictionary is created mapping the text of each directional input (i.e. 'left', 'right', …) to the selenium *Keys* object corresponding to that direction. Then, after the expectimax algorithm has been run on the reconstructed game board obtained from the screenshot, the key corresponding to the direction suggestion is sent to the webpage. The lines corresponding to these steps are shown below:

```
#Find the page's body element by its class name
game_board = driver.find_element(By.CLASS_NAME, "game-layout")

#Make dictionary for mapping predictor directions to output key
operations
move_dict = {'left':Keys.LEFT, 'right':Keys.RIGHT, 'up':Keys.UP,
'down':Keys.DOWN}
                                    ...

#Feed move direction back to webpage
    game_board.send_keys(move_dict[move_suggestion])
```

3.5.3 Additional Modifications

The agent is effectively complete at this point; however, some modifications can be made to aid in performance.

The easiest modification comes with the realization that the first several moves in any *2048* game don't affect the game in the long run, as there are very few tiles, all similar small values, meaning that they can be combined very easily. Additionally, the calculation of the first few moves of a game take a lot of time to complete, due to the start of the game having the most options for future game boards. Therefore, the first modification made to the agent was to make the first 8 moves all in random directions, with a 0.5s pause between each move to mimic the timings of the rest of the moves made. This leads to a less empty board by the time the computer vision and expectimax components come in, leading to more reasonable speed at the start of the game.

The next modification I've dubbed 'variable depth limiting'. This technique rests on the premise that if there are less moves to check the fuller a board is, and therefore a substantial increase in speed as the agent gets closer to losing the game (more tiles ~ closer to game end), the search depth of the expectimax algorithm can be increased in a controlled manner. As the expectimax algorithm search depth increases, two things happen: the expectimax algorithm takes longer and it produces a more well-informed prediction for the next move. This drop in expectimax speed is compensated for by the facts that a fuller board means less tiles need to be pre-processed via and that with less possible future boards to check the expectimax algorithm gains a bit of speed. Interestingly, the speed drop from a higher search depth and the speed gained from less required preprocessing and less possible future boards almost completely offset each other, leading to very smooth transitions between different search depths as the agent plays the game. The depth limit is changed depending on the value returned when the empty heuristic is applied to the current board, with decreasing empty heuristic scores indicating fewer empty tiles and thus a higher search depth, and vice versa.

The final small modification aims to create and store the board resulting from the move suggested by the expectimax algorithm without the new tile generated, which is then used to identify the empty tile indices in the previously discussed tile preprocessing process. With these three modifications, the agent is complete and successfully performs as well or better than the average human in both score and speed, all while playing entirely autonomously.

3.5.4 Agent Control Flow

Below is a flowchart illustrating the control flow of this agent. It is important to note that on the very first move of the game, there will be no board to use from the previous move, so instead a completely empty board is used as the previous move to force the computer vision component to check every single tile. Each subsequent move uses the previously generated board to aid in reducing the number of tiles necessary to pre-process and check.

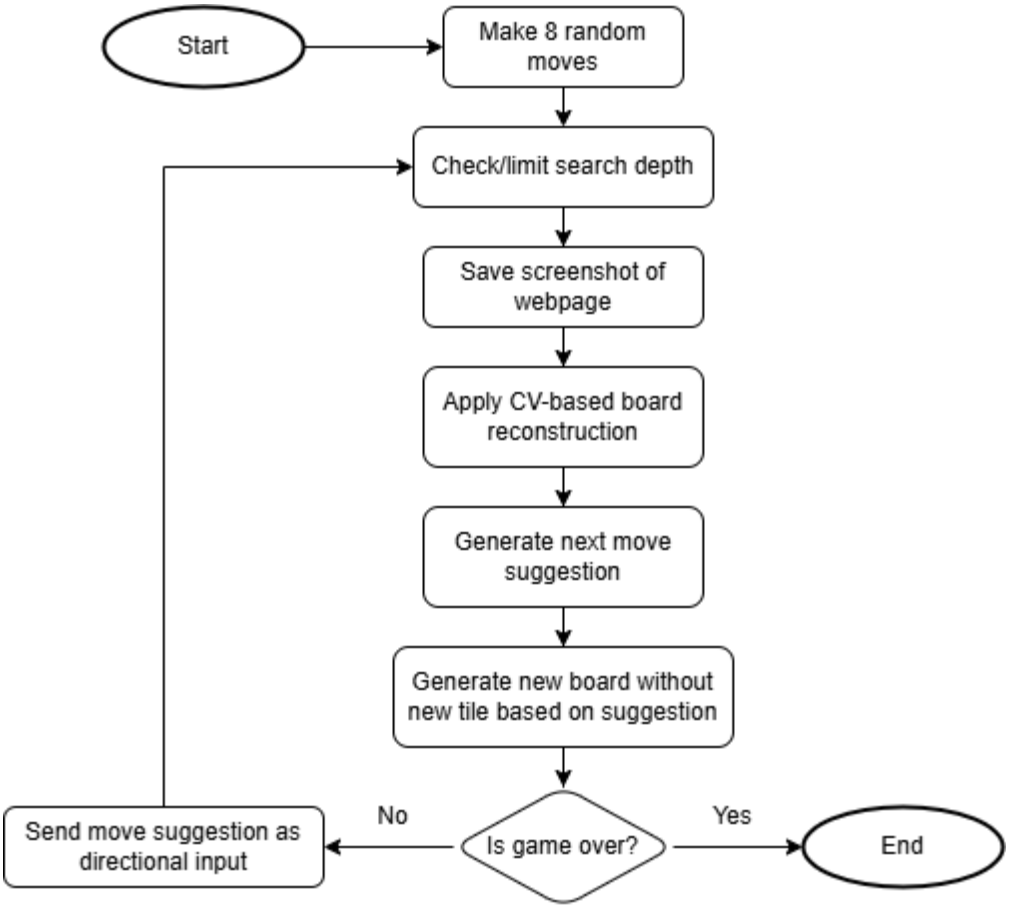


Figure 11. High level overview of autonomous agent control flow

Additionally, with the source code of this project complete, a UML class diagram can be composed showing the structure of the codebase, shown below:

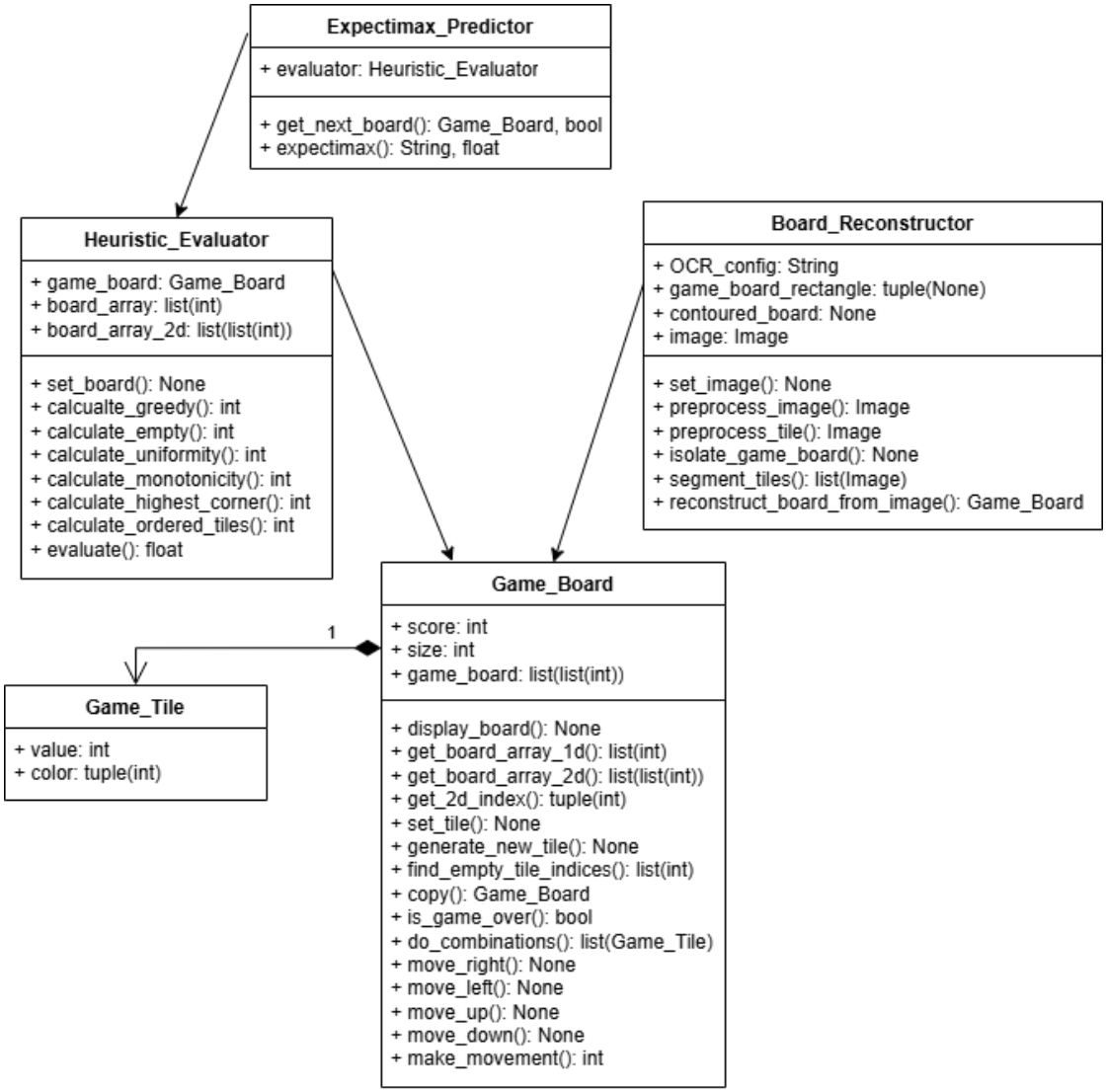


Figure 12. Class diagram of final codebase

It is important to note that the above class diagram does not include the code from the files entitled *2048_Window* or *2048_Agent*, neither of which are set up as a class but both of which utilize the classes shown in *Figure 12*.

3.5.5 Considerations

Upon completing both the computer vision and selenium components of this project, I realized that this process could be made both easier and more efficient by simply using selenium to reconstruct the board rather than using computer vision. This was originally not attempted due to a misguided idea at this project's conception that the tile values were not accessible via the HTML of the *2048* webpage, however upon closer inspection after completing the agent I found that the tile values are in fact accessible. However, although replacing the computer vision process with selenium would be much faster, I have stuck with computer vision for two reasons. Firstly, the process of developing the computer vision component of this project and learning to use technologies like OpenCV will be greatly beneficial to my future, as I plan to go into a field which uses these and similar technologies frequently. Secondly, although the selenium code I've written is specific to the *2048* official website, the computer vision component is not, and thus should work just as well on most other *2048* websites, providing adaptability which is simply not possible via the same process if using selenium for board reconstruction.

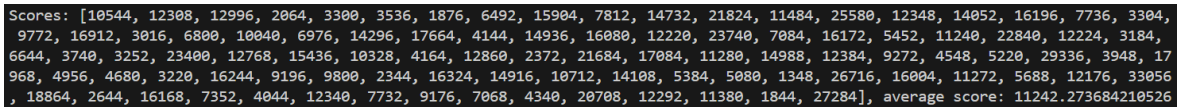
3.6 Testing Strategy

There are several components to the work done throughout this project, each of which was tested in a different way.

For the creation of the *2048* prototype, during development each of its components was tested through gameplay to ensure they worked properly. To aid this testing, an additional constructor was added to the Game_Board class which starts the game from a specified game board, which is fed through the constructor as an array. Using this, the functionality of each component of the game (e.g. tile movement, tile combination, random tile generation) could be tested by starting the game from a pre-determined start state where the result of the move taken is already known.

The largest tested component of the first term work was done in and around the heuristic evaluation. Since the heuristics are one of the most vital components in allowing the expectimax algorithm to function, they needed to be tested extensively. For this, two different methods were applied. The first of these was unit testing, applied individually to each heuristic method. These unit tests used heuristic evaluation on a pre-existing game board state where the outcome of the heuristic is known and the test suite included scenarios such as an entirely empty game board, an entirely full game board, a game board with complete uniformity, and several others.

The second testing method for the heuristics involved simultaneously testing the performance of the heuristics with the functionality of the expectimax algorithm. Since *2048* is a non-deterministic game, playing the game automatically will result in different max scores on each run. Therefore, to test the performance of the algorithm, the algorithm was left running for several iterations with each combination of different heuristic weights, and the average score across the iterations was used to assess the performance of that heuristic weight combination. Below is an image depicting the performance of the final heuristic weights:



The bulk of the testing in the second term was to do with the computer vision component, and was aimed at getting the OCR working properly on various test images. Therefore, most of the testing this term was done via a folder of screenshots taken manually of various *2048* webpages, game boards, and tiles.

Testing for the remaining components of the project was done by simply running the agent on the official website many times and recording if/when/why any errors occurred. This testing spawned both the variable depth limiting and empty tile checking concepts, providing the agent with both increased speed and increased accuracy.

3.7 Version Control System

With any large-scale programming project such as this, utilizing an appropriate version control system is vital. For this project, Git is used to ensure that changes are saved, backed up, and integrated seamlessly. Using Git also allows for the implementation of features on separate branches, allowing for different components of the project to be implemented without interfering with already working code.

Chapter 4: **Professional Issues**

As this project's focus is on the field of artificial intelligence (AI), it is important to investigate the effects this field has on society as its capabilities expand. Some such effects include the replacement of human actors in an increasingly online world with artificial agents, the impacts on our modern education systems affected by AI technologies like ChatGPT, and the effect AI has on the environment in both the present and future.

Due to how rapidly the field of AI is developing, it is difficult to know how emerging technologies will affect society in the short term, let alone the long term. However, it can be clearly seen from recent years that new developments in AI will have at least some short-term disruptive effects on society. This can already be seen to be happening in education, with Turnitin data from 2023 showing that at least 11% of assignments submitted to their system had been detected to have used AI in their writing [23]. Additionally, in my personal experience as a teaching assistant at Royal Holloway, I have witnessed or been made aware of students using AI in their assignments to a much larger degree, at a rate I would approximate near 50% (which is largely speculative, as this value has not been statistically verified in any way). The result of this use of AI can be seen firsthand in the capabilities of those students who rely on it to complete their work on their behalf, as often the students are left without an understanding of the content covered by the assignment they have just submitted. Conversely, if used in a supportive way rather than as a replacement for learning, AI could support a student's education rather than undermining it. For example, if a student lacks fluency in the language that their course is taught in or simply doesn't understand the content being taught in the way their educator delivers it, they could use some form of generative AI to provide a more personalized method of learning the content.

Additionally, educational institutions seem to struggle with appropriately responding to disruptive technologies such as ChatGPT, as the technology develops quicker than the bureaucracies surrounding these institutions can apply changes in response. This can be seen in the widespread responses to ChatGPT (and other generative large language models) taken by universities, with the technology initially being banned outright, only to have restrictions loosened or removed completely 2-3 years later. It is my view that regardless of how oversold these emerging AI technologies are, as detailed in Niel Selwyn's paper on the future of AI and education [24], their current capabilities are such that they will likely be used for the foreseeable future, meaning that to properly prepare students educational institutions should be teaching the use of these technologies as a tool (like Google search or Microsoft Word) rather than restricting their use. It is important to note however, that as pointed out in Selwyn's paper, today's AI has flaws (such as providing outright incorrect responses or deepening the biases present in the data the models were trained on), yet if educational institutions were capable of keeping pace with emerging technologies like this, students could be taught to identify and work around said flaws.

While the current effects of AI are visible to some degree, it is likely that the future effects will far outweigh anything happening now. In any industry which is computerizable, nearly 100% of the human workforce could potentially be replaced by AI agents. For any task ranging from formatting spreadsheets to frontend web development, we are already seeing AI programs emerge which can completely automate the process. Additionally, with the aid of robotics, simple manual labor jobs could also be vulnerable to replacement by AI, for example the assembly of automobiles which could be mostly or entirely delegated to AI-powered robotics. These are roles which currently and previously have been filled by humans, meaning that as the workforce becomes increasingly composed of autonomous agents, an increasing number of people will lose their jobs as they are replaced. This potentially leads to a widespread unemployment crisis at the behest of AI development. It is important to note, however, that as stated in a speculative Pew research center report on the impact of AI on the future of jobs [25] that technological developments tend to displace jobs in certain industries in the short term, however overall technological developments are net creators of jobs. This suggests an inverted u-shaped curve would be the most appropriate

model of the effect of AI on the workforce over time, with large-scale disruptions at the start, eventually flipping to create more jobs than were lost in the first place.

A similar inverted u-shaped curve can be seen with the predicted effect of AI on the environment, with experts believing that in the short term the technology used for AI will be a massive emitter of carbon emissions (as can already be seen to be happening with the high energy demand caused by the training and use of LLMs [26]), but in the long term the innovation driven by the use of AI will lead to a net decrease in emissions, as seen with the effect on the workforce.

Therefore, the development of artificial intelligence is likely (and can already be seen) to cause some major disruptions to society, however over in the long term it will likely result in a net-positive impact on society. This is, however, if development of AI is done in a manner conscientious of the potential long-term negative impacts. If these impacts are ignored, the future development of AI could cause irreparable damage to the environment and our society. Contrarily, if the environmental, economic, and educational impacts are considered at all stages in the development of this new technology, then it has the potential to markedly improve the life of the average human.

Chapter 5: **Reflection**

Upon completion of this project, I can confidently say that I have produced a final product that I am proud of. I achieved all the goals I laid out for myself at the beginning of the year, and developed extensive skills and knowledge I would not have otherwise gained. For example, I am now capable of using both the Selenium and OpenCV libraries competently, the former of which I have always felt necessary to learn and the latter being an important technology used in the field I intend to join for my career.

Parts of this project could have been improved, such as the replacement of the entire computer vision component with a more efficient method of game board reconstruction via Selenium, however this change has both pros and cons, and overall, I feel the choices made during the development of this project have been reasonable. The clearest area where I could have improved the development of this project is my time management, via both the initial planning and the creation of the project. When compiling my planned project timeline, I greatly overestimated the time it would take me to complete the code (as I tend to work much more diligently and efficiently when working on things I enjoy, such as the programming involved in this project) and as a result the time allotted for completing the report could have been greatly expanded.

All in all, though, this project was an excellent learning experience for both programming and report writing, both of which will be vital for my future career in software development.

Bibliography

[1] Cirulli, G. (2014) *Gabrielecirulli/2048: The source code for 2048, GitHub*. Available at: https://github.com/gabrielecirulli/2048.

[2] Selenium Community (2023) *The selenium browser automation project, Selenium*. Available at: https://www.selenium.dev/documentation/.

[3] OpenCV (2024) *OpenCV modules*. Available at: https://docs.opencv.org/4.10.0/.

[4] Chess.com Team (2018) *Kasparov vs. Deep Blue: The match that changed history, Chess.com*. Available at: https://www.chess.com/article/view/deep-blue-kasparov-chess.

[5] BBC (2016) *Artificial Intelligence: Google's alphago beats go master Lee Se-dol, BBC News*. Available at: https://www.bbc.co.uk/news/technology-35785875.

[6] Russell, S., Parks, A., Kautz, H. and Shapiro, L. (n.d.). *Mausam (Based on slides of.* [online] Available at: https://courses.cs.washington.edu/courses/cse573/12sp/lectures/24-games.pdf.

[7] Kanade, V. (2022). *What is Heuristics? Definition, Working, and Examples*. [online] Spiceworks. Available at: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-heuristics/.

[8] Goel, B. (2017) *Mathematical analysis of 2048, the game*. Available at: https://www.ripublication.com/aama17/aamav12n1_01.pdf.

[9] Hoang, L.N. (2016) *The addictive mathematics of the 2048 tile game, Science4All*. Available at: https://www.science4all.org/article/2048-game/.

[10] IBM (2021). *What is Computer Vision?* [online] Ibm.com. Available at: https://www.ibm.com/think/topics/computer-vision.

[11] OpenAI (2024). *ChatGPT*. [online] Openai.com. Available at: https://openai.com/chatgpt/overview/.

[12] Amazon (2025). *What is OCR (Optical Character Recognition)? - AWS*. [online] Amazon Web Services, Inc. Available at: https://aws.amazon.com/what-is/ocr/.

[13] TensorFlow Datasets (2010). *mnist | TensorFlow Datasets*. [online] TensorFlow. Available at: https://www.tensorflow.org/datasets/catalog/mnist.

[14] tesseract-ocr (2024). *GitHub - tesseract-ocr/tesseract: Tesseract Open Source OCR Engine (main repository)*. [online] GitHub. Available at: https://github.com/tesseract-ocr/tesseract?tab=readme-ov-file.

[15] Lee, M. (2022). *pytesseract: Python-tesseract is a python wrapper for Google's Tesseract-OCR*. [online] PyPI. Available at: https://pypi.org/project/pytesseract/.

[16] pptr.dev. (2025). *What is Puppeteer? | Puppeteer*. [online] Available at: https://pptr.dev/guides/what-is-puppeteer.

[17] Playwright.dev. (2025). *Library | Playwright*. [online] Available at: https://playwright.dev/docs/library.

[18] Shain, D. (2024). *Playwright vs Selenium: What are the Main Differences and Which is Better?* [online] Automated Visual Testing | Applitools. Available at: https://applitools.com/blog/playwright-vs-selenium/.

[19] Pygame (n.d.). *Pygame Front Page — pygame v2.0.0.dev15 documentation*. [online] www.pygame.org. Available at: https://www.pygame.org/docs/.

[20] Kohler, I., Migler, T. and Foaad Khosmood (2019). Composition of basic heuristics for the game 2048. doi:https://doi.org/10.1145/3337722.3341838.

[21] www.ncl.ac.uk. (n.d.). *Numeracy, Maths and Statistics - Academic Skills Kit*. [online] Available at: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/descriptive-statistics/expected-values.html.

[22] Restack.io. (2025). *Python Recognize Numbers In Image | Restackio*. [online] Available at: https://www.restack.io/p/image-recognition-answer-python-recognize-numbers-in-image-cat-ai.

[23] Turnitin.com. (2024). *Turnitin marks one year anniversary of its AI writing detector with millions of papers reviewed globally*. [online] Available at: https://www.turnitin.com/press/turnitin-first-anniversary-ai-writing-detector.

[24] Selwyn, N. (2022). The Future of AI and education: Some Cautionary Notes. *European Journal of Education*, [online] 57(4). doi:https://doi.org/10.1111/ejed.12532.

[25] Smith, A. and Anderson, J. (2014). *AI, Robotics, and the Future of Jobs*. [online] Pew Research Center: Internet, Science & Tech. Available at: https://www.pewresearch.org/internet/2014/08/06/future-of-jobs/.

[26] Mehta, S. (2024). *How Much Energy Do LLMs Consume? Unveiling the Power Behind AI*. [online] Association of Data Scientists. Available at: https://adasci.org/how-much-energy-do-llms-consume-unveiling-the-power-behind-ai/.

Appendices

Appendix 1: Link to demo video

- https://youtu.be/c3gskRvdc30

Appendix 2: User Manual

To run the programs entailed in this report, a user needs to:

- Import code (either to an IDE or to a folder)
- Ensure that Python 3 is installed, as well as the following packages:
    - numpy
    - pygame
    - pytesseract
    - cv2
    - selenium
- Run the code from the appropriate entry point:
    - If the user wants to use the final autonomous agent, run the 2048_Agent.py file
    - If the user wants to use the local expectimax agent from term 1, run the 2048_Window file
        - The search depth in this file can be changed by changing the value on line 21. This value should only be multiples of 2, and values greater than or equal to 6 take extremely long to run.

If the user also aims to run the small Testing.py file, they should also:

- Install the pytest package
- Run the Testing.py file

Appendix 3: Project Diary

FYP Work Diary

30 September 2024

- Organized and attended first student-supervisor meeting

- Conducted research into the following topics:
    - Which game I plan to use for my project (decided on the tile game 2048)
    - Previous approaches taken to use AI to beat 2048 (i.e. expectimax algorithm, reinforcement learning)
    - Computer vision algorithms and libraries to be used to automatically play game

01 October 2024

- Expanded project timeline plan, including a rough outline of steps for how the programming process will unfold
- Conducted more thorough research into the formatting of the plan report, interim report, and final report
- Conducted research into the following topics:
    - Selenium
    - LaTeX
    - Previous applications of computer vision for playing video games

02 October 2024

- Emailed Ilia my timeline outline to see if I am currently on the right track
- Developed report writing timeline further, including dates from Moodle
- Added source for minimax algorithm, to be looked into further in future

03 October 2024

- Created document to be used for project plan
- Started work on abstract, getting most of the description of 2048 written up
- Created a week by week timeline for term 1
- Got feedback from Ilia that I do seem to be on the right track
- Added sources for fundamentals of CSP and pedagogical possibilities of 2048

04-05 October 2024

- Expanded abstract to start talking about adversarial search problem approaches
- Started a list of potential risks, to be expanded upon

06 October 2024

- Finished abstract draft, needs to be revised in future (make game description more concise, make structure more cohesive, clarify impact of project, and remove excessive examples)

- Added to list of potential risks, now need to convert list into full descriptions of said risks, along with their potential mitigations

07-09 October 2024

- Revised and finalized abstract, including adding sources in text.
- Added references section containing all resources used for development of plan document.
- Wrote out risks and mitigations section, detailing the risk posed by each potential issue and their associated mitigations.
- Polished and completed plan document, then submitted it to moodle.

10 October 2024

- Started work on making 2048 prototype in Python using Pygame
- Made a window with a 4x4 grid drawn on it, with no additional functionality as of yet
- Issue with grid lines leaving a small gap in the top left corner of the grid; not an issue for future gameplay but will potentially be annoying visually

11 October 2024

- Created a class for the game board which will contain the individual tiles
- Created a class for the game tiles to be used in the game board
- Added visual drawing of the game board and game tile onto the Pygame window, providing the ability to see a visual representation of the game
- Added movement for tiles in each of the cardinal directions, by going row by row (column by column for up or down), extracting out the non-zero tiles, then re-adding tiles of value zero to whichever side the tiles moved from (0 2 0 4) -> (2 4) -> either (0 0 2 4) or (2 4 0 0) for right and left respectively
- Added tile combination during movement

12 October 2024

- Reworked movement functionality to return a value which confirms whether any movement in the requested direction actually occurred (if a right input, returns whether or not anything was actually able to move to the right)
- Implemented random new tile placement after player makes a move, by generating a random grid index then iterating it until an empty space is found, stopping at 16 attempts as at 16 attempts without a successful placement the grid is full
- Added score calculation and displaying
- Added detection for when the game ends (i.e. gives a Boolean true value if attempts for left, right, up, and down all fail)
- Added a game over screen, displayed when the game end detection returns true

16 - 20 October 2024

- Started research into heuristic analysis, minimax algorithms, game tree pruning, expectimax algorithms

- Compiled a top level plan on how exactly to implement expectimax solver, including how heuristics will work

21-25 October 2024

- Created new branch for making and testing heuristics for game board evaluation

- Made a class to act as a heuristic evaluator to be applied to a game board

- Implemented 4 heuristic calculators within the heuristic evaluator:

    - Greedy heuristic which sums the total tile values on the game board

    - Empty heuristic which returns the number of empty tiles on the game board

    - Uniformity heuristic which counts the number of tiles who share the same value

    - Monotonicity heuristic, which goes through the game board by row and column and checks whether all the tiles in each row/column are either non-increasing or non-decreasing, and gives a percentage score as a result to represent how monotonous the board is. The percentage is calculated by 1 – (number of non-conforming tiles / number of non-empty tiles)

- Implemented first uses of TDD within project, used for checking that heuristic calculations yield the expected results

28 October 2024

- Tweaked heuristic calculations to return normalized results in the range [0, 10], to allow for them to be used in combination with each other and weighted

- Added a general heuristic calculation which uses each of the 4 heuristic calculation methods to provide a general score for the game board

- By just using the score provided by each calculation method, worse boards were scoring the same or better than better boards, so tried weighting greedy calculation to 2x its value, empty and uniformity to 0.5x their values, and leaving 1x monotonicity, which provided more accurate values for the test boards I used. Need to test heuristics with many other boards to see how effective they are currently, but apart from weighting the components of the evaluation differently the heuristic evaluator should now be complete

- Used Bhargavi Goel's paper to determine the max board score for the greedy analysis

04 November 2024

- Created class to start expectimax algorithm implementation

- Add a method to class which generates a child board and returns whether this generation was successful

- Moved game over method to Game_Board class from gameplay loop so that it can be accessed by each game board

- Start making expectimax algorithm by implementing player move steps

06-10 November 2024

- Completed expectimax algorithm implementation
- Fixed a bug where the method that generates and checks new boards was always returning None
- Tested algorithm at low depths
- Added an additional Boolean heuristic that gives a large reward if and only if the largest tile is in a corner
- Added a button to GUI which shows expectimax evaluation of current game board
- Upon confirmation that expectimax appeared to be working as it should be, made 2048 prototype take the move returned by the expectimax analysis automatically, then preform the analysis again on the new board, such that the game plays automatically
- Performance already surprisingly effective
- Added a massive punishment (evaluation = 0) for boards whose game has ended

11 November 2024

- Changed expectimax prediction button to a game reset button
- Fixed bug where 1d game board representation generation was instead giving 2d representation
- Made 2048 prototype automatically reset game to a new game after the expectimax bot loses

15-20 November 2024

- Experimented with search depths and heuristic weights to find ideal weighting of heuristics
- Added another Boolean heuristic which rewards if and only if the largest (up to) 4 tiles are in the same row

02-07 December 2024

- Create and upload presentation for interim review
- Begin work compiling interim report

08-12 December 2024

- Continued and finished work on interim report, compiling background theory, current project progress, and project aims into a well-structured, complete report
- Created demo video and added to interim report
- Submitted interim report

13 December 2024

- Presented interim report presentation

Term 2

22 January 2025

- Started compiling notes on how exactly I plan to detect and reconstruct a 2048 game board from a website
- For now, best option seems to be selenium for webpage screenshot, OpenCV for image preprocessing and tile contouring (for reconstructing the correct position of the tiles), and tesseract library for easy digit recognition

25 January 2025

- Started prototyping the image recognition technology which will be used to identify the game board and the tile values
- Had an issue where digits are successfully recognized on a 2048 webpage screenshot except those on the game board, which are the only digits I need, even after preprocessing via gaussian blur and Otsu thresholding
- When the screenshot is cropped down to a single tile on its own (i.e. an image only containing a '4' tile), after preprocessing the OCR is flawless, therefore the solution to the above problem will be to do tile contouring first to find tile locations, then individually extracting 1 tile at a time (potentially via image slicing, but this is yet undetermined) and using OCR on each of them

27-28 January 2025

- Added initial contouring process, which works well except for 16, 32, and 64 tiles
- 16, 32, and 64 tiles blend in with background when preprocessed to a binary image, causing them to be un-contourable

01 February 2025

- On a visit home to family, my uncle, who is an ex-Google employee, discussed with me potential solutions to the problem of 16, 32, and 64 tiles not being contoured
- Although he isn't familiar with modern computer vision techniques, rather principles from ~25 years ago, he suggested an idea which I had already thought of and passed off as a last resort: contouring only the game board then manually segmenting into individual tiles
- If I can't adjust the preprocessing to distinguish these tiles from the background, this idea will be the next solution
- Pros: clearer implementation steps
- Cons: likely will have slower runtime due to needing to apply digit recognition to each individual tile (although this was potentially necessary with the tile-contouring idea anyway)

05 February 2025

- Attempted to "greenscale" image rather than greyscale to try to identify only the missing tiles then overlay contours onto existing tile contours, however, could not manage to "greenscale"
- Attempted using adaptive thresholding rather than Otsu's thresholding for binarization, which ended up being less effective than Otsu's thresholding and didn't solve the problem at the right step in preprocessing (needed to implement a change before converting to greyscale, as the greyscaling process seems to be where the tiles become indistinct from their background)

- Adjusted contouring boundaries to only accept contours whose width and height are less than 100px different, and which have a width greater than 300px (to ignore all tiny square-like contours), as well as ensuring that x position is greater than 10px (to avoid including a contour around the entire image
- This strategy manages to successfully isolate 1 contour around only the game board, which fits the board extremely well
- Using this contour, array slicing is used to isolate the game board as a new image, which I tested with different full-page screenshots, and it appears to work well
- This will be used to isolate the individual tiles by dividing the new image into a 4x4 grid

11 February 2025

- Applied preprocessing to each tile to be identified individually, however was still missing some detection for certain tile cases
- Resized tile images to be 500px by 500px, then only processed the center 200px by 200px square, which allowed for successful processing of '2' and '4' tiles, however a new issue arose with longer-valued tiles not being able to be detected
- Possible solutions:
    - The only new tiles which will generate are '2' and '4' tiles, so with reliable detection of their positions, I can manually determine the result of a directional movement before the tile has been generated, then look through the image processing and only add the newly generated tile (done by going tile by tile with the movement-without-new-tile board and the detected board, and setting the tile value to either '2' or '4' if the tile is different from the movement board, while prioritizing the movement board. This would possibly result in more tiles being placed than there really are if the OCR falsely detects a '2' or a '4' on an empty space)
    - Potentially instead of resizing and cropping the tile images, preprocess them to a binary image, then use the findContours method and isolate the digits that way, then send them through OCR (could potentially be more reliable than the current method)

13-14 February 2025

- Attempted to solve previous issue by first trying to contour the text (digits) in each preprocessed tile image, however the contouring didn't work nearly as expected and ended up making the issue worse. Likely could have made this work if I probed further into it, however a different solution ended up working better:
- With a square-cropped version of the tile images, sometimes a corner of the tile border would be left in the image
- Since the image was previously binarized, this corner becomes a problem, as if the tile is empty, like the one shown above, pytesseract will attempt to identify the only discernable feature (the bottom left corner) as a digit, and in the case of the above image it identifies it as a 7. This is a problem as it introduces 'false positive' digits in place of what should be empty tiles, but this same issue is what leads to the missing of other tiles
- The solution to this came when I realized that every single tile, no matter its value, will always only need to be 1 digit's height tall. The tile values may vary in length depending on the number they are displaying, but since the digits on the tile never extend to multiple rows, the tile image can be cropped to be much larger horizontally than vertically
- By making the tile cropping use the center 400px by 220px, any additional noise introduced by the tile borders is completely removed, and remarkably this one change makes the tile-digit-detection flawless. Having introduced this change, my program can

now reliably identify the tiles in a game board from an image and successfully reconstruct them into a Game Board object.
- Next step is to set up automatic webpage screenshotting to feed to the OCR program, which can then be used to perform many more tests of the OCR program in quick succession, while simultaneously effectively completing the code portion of this project

15 February 2025

- Started working on developing Selenium-based webpage screenshotting, successfully managing to visit the webpage, take a screenshot, save it to and override previously saved screenshots in a new img//selenium_img folder, then close webdriver
- Had an issue where when opening up a new 2048 webpage, cookies need to be accepted before the game can be played, which I solved by simply searching the page for clickable elements with the expected text in them then clicking the button, which works quickly and reliably. Had one failed attempt to solve this issue by trying to identify buttons from their html class/id, but could not find the cookie popup section in the html doc structure

16 February 2025

- Converted OpenCV based game board detection file into a class-based OOP structure, to be used for connecting with the expectimax predictor and selenium components
- Connected all components together, and got them working to successfully play 2048 completely autonomously
- However, several issues arose (some have been fixed, and some are yet to be explored):
    - The computer vision component sometimes misidentifies the 32 and 512 tiles, with 32 sometimes misidentified as either 52 or 352, and 512 as 912. This was solved by adding a case to the Game_Tile class which manually converts the tile value to 32 or 512 depending on if the given tile value is one of the previously mentioned problematic ones. Further game testing needs to be conducted to identify if other tiles are misidentified. If this issue persists with non-static values, could add a case to the gameplay loop where if a tile is identified to be invalid, a random move is taken, and a new screenshot is produced
    - The program is extremely slow. It takes around 1.5s to make a move on the lowest expectimax search depth, which makes the game play much slower than the average player (although it accomplishes more given a long enough timeframe). I attempted to combat this issue by reducing the number of times preprocessing is applied to the individual tiles, however any adjustments made to the individual tile preprocessing caused extreme inaccuracy in the computer vision's OCR.
- After having used selenium and webdriver for the input and output components of the agent, I could possibly identify the tile values from the html doc structure instead of using computer vision, which would cause tons of work to be useless, but would potentially be much faster and more accurate
- Need to schedule a meeting with Ilia to discuss

14 March 2025

- After my previous meeting with Ilia my code has been deemed good enough to consider finished, so work has moved to compiling the final report
- Laid out subheadings for documentation of term 2 background theory and resulting code

23 March 2025

- Moved tesseract folder from my local machine to the project folder and configured to work like that, so that anyone running my code shouldn't have to download tesseract

- Started writing technical documentation of term 2 work, detailing the process and design of the computer vision component responsible for board reconstruction (except for a final subheading entailing optimizations applied to this component and how the parts of the component came together)

31 March 2025

- Had my final meeting with Ilia, discussed layout of final report document and determined it is mostly okay, but need to plan to add some UML diagrams to the software engineering section

04 April 2025

- Finished writing up the computer vision component section of final report
- Started and finished writing up selenium component section of final report, including a flow chart entailing the control flow of the completed autonomous agent, leaving only term 2 testing strategies and additional UML diagrams (i.e. class diagram) unfinished in the software engineering chapter of the report

05 April 2025

- Finished testing strategy section of report
- Finished Selenium background theory section of report
- Laid out bullet points for computer vision background theory
- Filled in section on why this project helps me
- Went through entire introduction chapter and abstract and touched up for final report

06-08 April 2025

- Finished draft for all chapters
- Added a small 5th chapter reflecting on the project
- Completed and updated references
- All that is left for final report complete draft is:
    - Update demo video
    - Final polish on code, move to main branch
    - Update appendix to include 2nd term work diary
    - Add 2nd term work diary to diary.md
    - Update readme
    - Add short user manual to appendix detailing how to get program running on any machine

09 April 2025

- Complete final report, polish remaining loose ends