Queen Elizabeth's Grammar School

AQA Computer Science NEA

Jabir Hussain

Candidate No: 3037

Centre No: 26324

Department of Computer Science

April 17, 2020

Contents

Chapter 1

Analysis

1.1 Introduction

Redacted is the Head of the Computer Science (CS) department at Queen Elizabeth's Grammar School. Computer Science has always been of interest to him, but his past experiences have been based on software development for clientele. Because of this, he has had little chance to explore fields of theoretical computer science, namely the basics of artificial intelligence and the subfields which it comprises of. To discuss this further, I interviewed him to gain a better idea of his current understanding of this topic. [1]

1.1.1 Interview

- **"So what aspects of Artficial Intelligence interest you in particular?"**
  - "Machine Learning, Evolutionary algorithms and neural networks. Though to be honest, I know little about them. That being said, I'd love to know more"
- **"What type of product would provide you with a more dynamic understanding of Artificial Intelligence: a data analytics application, or an interactive game?"**
  - "Both probably. One way of learning would reinforce the other. A data analytics project would give me a feel for the complexity of the topics, however a game would provide an engaging form of learning. With that in mind, there's probably some way to incorporate the two in a single project."
- **"What sort of devices would the application run on?"**
  - "I'm currently using a Microsoft Surface Pro, which is quite a capable machine. That being said, the students in the school can't bring their own devices in, so the project cannot be too resource-intensive. There's no chance an AAA game would run on these computers, nor could they run Big Data analysis applications!"

[1]Note: bold text is from my perspective

- **"You've mentioned students - do you intend on using a this as a teaching tool? Or an application to provide an example of decision-making algorithms?"**
  - "This wouldn't be used as a teaching aid - the program should be an easy-to-follow application, alongside brief descriptions of the algorithms."
- **"Any further comments?"**
  - "In terms of the user interface, the choice is yours. Computationally competent students will be using this project - using a CLI would give the students a chance to build their confidence in using a terminal, however a GUI will provide a smoother user experience."

Whilst reading the AQA NEA Guideline, it suggested the use of Tic-Tac-Toe as a means of investigating machine learning algorithms [**5**]. Tic-Tac-Toe is a game where players attempt to get three symbols in a row on a 3 by 3 grid, taking turns when placing their symbol down. It is regarded as a "solved game," meaning every possible outcome has been researched and identified. It is a game that the majority of people are familiar with, in which machine learning and decision making algorithms can be implemented as an opponent.

1.1.2 Identification of Prospective User(s)

There are two groups of people who will primarily use this application: the Computer Science department staff, and avid students studying Computer Science. The users will be computationally proficient.

1.1.3 Objectives

- Implement an independent machine learning algorithm for a 3 by 3 game of Tic-Tac-Toe.
  - The algorithm must have no previous knowledge of the game, and gradually learn how to play the game effectively over a set number of games.
  - They should progressively get better as it learns
- Implement a decision-making algorithm to play a game of Tic-Tac-Toe
  - The algorithm must be able to play a perfect game of Tic-Tac-Toe
  - It must have an optimisation to aid its game analysis
- The algorithms must have flexibility so users can amend some settings.
- A graphical user interface:
  - An easy-to-use interface must be provided to the player, giving relevant descriptions to the player about how the algorithm works.
- Data must be recorded in a suitable fashion, through the use of complex data structures and/or databases

1.1.4 Data Handling

The majority of Artificial Intelligence algorithms have a number of constants which need to be programmed, such as learning gradients, number of recursions, etc. To ease the modification of these constants, I can either use a database or a built-in data structure. Using a relational database would be unnecessary - my data would be in atomic form, and the data doesn't follow a typical "Entity-Attribute" model, which is an underlying principle for the use of databases. Nevertheless, it would be an additional strain on resources to implement. On the other hand, using a dictionary provides clarity on the algorithm the constant is used for: for accessing constants, it uses more memory but it would provide faster querying. This means recalling constants in the program will be faster than using an external data file, but for a piece of text which is only accessed once, such as the description of the algorithms, using a JSON string format would be easiest.

Python has a number of JSON parsing libraries, but I decided to use the "ujson" (short for UltraJSON) library. Upon research, I learned that the module is 3.25 (to 2 decimal places) times faster than the preinstalled JSON package that comes with stock Python [**3**].

1.2 Research

The fact that the game is solved means we could implement a brute force algorithm which iterates through every possible outcome, but with 255168 possible outcomes on a mere 3 by 3 grid, iteration and comparison would be an ineffective solution, demonstrating the need to research decision-making algorithms. For decision-making algorithms, a 3 by 3 grid could pose as a simple challenge, so I intend to program a 5 by 5 implementation alongside the 3 by 3. The game is simplistic as opposed to a traditional board game like chess, and the state space is considerably smaller. In this context, the state space could be referred to as a set of every possible board configuration for the game. The state space affects the time and space complexities of the algorithms, so I researched into the influence it has on decision-making algorithms, and how I can reduce it. Every square can hold 3 values, more appropriately referred to as states. It can either be a cross, nought or a blank space. By inference, we can deduce that there is a total of 3^9 outcomes, but this calculation includes boards that are filled with noughts, or boards with multiple 3 by 3 patterns. Researcher Brian Shroud devised a size function, providing an upper bound for the possible outcomes. It returns an upper bound because it still considers states which have both players winning, but it reduces the number of outcomes by a factor of 3. The size is given through the following calculation:

$$S(n) = \sum_{r=0}^{n} \frac{n!}{(n-r)! \left\lfloor \frac{r}{2} \right\rfloor! \left\lceil \frac{r}{2} \right\rceil!} \tag{1.1}$$

where n is the area of the grid. Using this function, a 3x3 grid yields a "size" of 6046, as opposed to the initial 19683 with the iterative approach. We should also consider symmetrical state spaces:

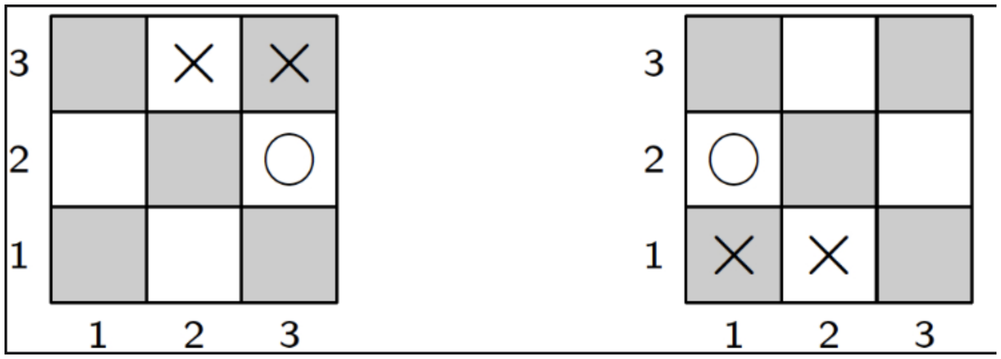


Figure 1.1: *Example of symmetry in a board*

As we can see in Figure 1.1, the two grids are the same, just different points of rotation. By neglecting similar boards, we reduce the state space further, resulting in a faster algorithmic analysis. There are a number of decision-making algorithms used for game AI's, but I have narrowed it down to three.

1.3 Algorithmic Analysis

1.3.1 Reinforcement Learning (RL)

Reinforcement Learning is a form of Machine Learning based on incentivising; the machine is presented with a "reward" when it is learning as intended, and a "punishment" when it isn't performing the task properly. Specific to RL, the variable providing the change is called the **Agent**, and in the case of this NEA, the player is the Agent. The reward is winning a game, as the player intends to win every game possible. Reinforcement Learning is a broad branch of Machine Learning, so for my coursework I will focus specifically on Q-Learning. Q-Learning is a specific type of Reinforcement Learning with the following criteria (it should be noted that the definitions of the terminology are only for Q-Learning):

- **Learning:** We do not program the machine with a particular skillset or approach to a game, as it learns itself
- **An Environment:** The "black box" region the Agent is exposed to, where they learn.
- **An Interaction:** What the machine advises itself when it is in an unidentified environment
- **The Policy:** To maximise the rewards the machine gets, the Policy gives the machine the most effective Interaction relative to the state in the Environment

Q-Learning is table-based learning, through the use of a "Q-Table." The table provides values of "Actions" against "States," where it has numerical values that represent the reward. This reward value is determined by the "Q-Function:"

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot Q'(s_t, a_t) + \alpha(r_t + \gamma \cdot \max Q(s_{t+1}, a_t)) \tag{1.2}$$

Where:

- s_t is the current state
- a_t is the move that the machine executes
- s_{t+1} is the following state
- r_t is the reward the machine receives after a_t
- $Q'(s_t, a_t)$ is the previous value
- α is the learning gradient; it is the variable that controls how much old information is overwritten. For some α value n, the agent neglects the previous learning for n number of cycles and overwrites it with the new learning it undergoes. For an α-value of 0, the Agent is able to use all previous knowledge. The values are typically 1 or 0, so we can model this as a step function.
- r_t is the reward value
- γ is the discount factor; this is the numerical value that determines the significance of potential rewards. If the γ-value is 0, the machine is regarded as "short-sighted." It can only refer to previously achieved rewards, and the move will seem beneficial when it could actually provide the opponent with an opening to win the game (i.e. $\gamma = 0$.) With a higher value, the machine looks and calculates for a long-term reward.

There are two Q-Learning strategies: exploitation and exploration. As it suggests, exploitation involves the use of previous knowledge by Q-Table reference to choose the most effective move (with the largest reward). On the other hand, Exploration is a different approach. The machine randomly plays moves in hopes of finding a higher reward than the ones it already has. But where do these rewards come from? We train the machines with data, data it provides itself. The machine plays against itself using both exploitation and exploration, adding values to the Q-Table every epoch.

Reinforcement Learning is one of the optimal solutions for me: it is beginner friendly, with an abundance of documentation online and in Public Libraries, with very little mathematical depth as opposed to the use of an Artificial Neural Network (ANN), which uses multivariate calculus for its backpropagation algorithm:

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i}\frac{\partial E}{dy_i}, \tag{1.3}$$

which requires a significant amount of processing power to run efficiently and effectively. It is evident that the Q-Function is an easier implementation. Nevertheless, with the use of the variables in the Q-Function, I have the flexibility to adjust whether I want "short-sighted" or long-term learning, or to find the medium. In the instance of Tic-Tac-Toe, the machine will need to consider the short-term to evaluate how its decision will affect the next move of the opponent, but it must also consider the long term goal of winning the game. It is also a machine learning technique ideal for our environment; the Agent needs to maximise its reward, similar to how human players want to maximise their number of wins. [2]

[2]But remember, it's taking part that counts!

1.3.2 The Minimax Algorithm

The minimax algorithm is one of the most famous decision-making algorithms, having been implemented in game engine software such as StockFish - one of the world's most renowned chess engines. Simply put, the main objective of minimax is to reduce the opponents reward. An evaluatory function calculates a score, ranking the best possible moves. Minimax uses recursion to explore every possible move, allowing the machine to find the optimal move. The evaluatory function has a positive score for when the machine wins, and a negative for when the opponent wins. There are variations of minimax with different emphasises on the reward schematic. One variation finds positive and negative scores close in absolute value, to try and minimise the opponents opportunity to have a winning move whilst having a winning move themselves; this zero-sum approach to the heuristic algorithm is more commonly known as the Negamax algorithm. Minimax is an ideal algorithm, as it evaluates every possible move and chooses the move which will maximise the reward; by evaluating every move, it should be a perfect algorithm. But the standing issue is the complexity - as the dimensions of the board grows, so does the size, and the algorithm ends up sifting through hundreds of gigabytes of data. Because of this, Minimax has complexity $O(N^d)$, a polynomial complexity. But as N, the number of possible moves, grows and d, the depth that algorithm searches, grows, it begins to approximate to an exponential relationship. We can counteract this by implementing further algorithms: I have chosen to implement an alpha-beta pruning method. Alpha-Beta pruning stops the minimax's evaluatory function when a better move has been previously found, and moves onto the next evaluation.

1.3.3 Monte-Carlo Tree Search (MCTS) Algorithm

Monte-Carlo Tree Searching is different to the evaluations the other algorithms conduct; MCTS sets states as nodes, and traverses this tree to find the ideal move. It uses a back-propagation method, where it tests one state and looks at the outcome of the game from that state transition. The outcome is then implemented in the other nodes, allowing us to compare every state following that move to see whether it is viable. We implement this by choosing some root node, and plotting the child nodes accordingly. If the leaf node isn't a terminal state, append a node to the tree which is the next move, and with this new node, simulate the outcome of the game until the game finishes. This process is repeated until a terminal state is reached. The nodes have a weighting, with the largest score being the best move to make. This score is given using:

$$Score(X) = (v_i + C) \cdot \sqrt{\frac{\ln N}{n_i}} \tag{1.4}$$

where:

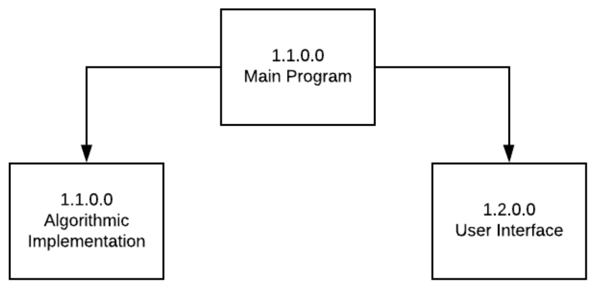
- v_i the average win count
- C is the bias; MCTS uses both exploration and exploitation, and the bias determines which has a greater emphasis.
- N denotes the visit count of the parental node, with n_i being the visit count of the node i away from the parent node.

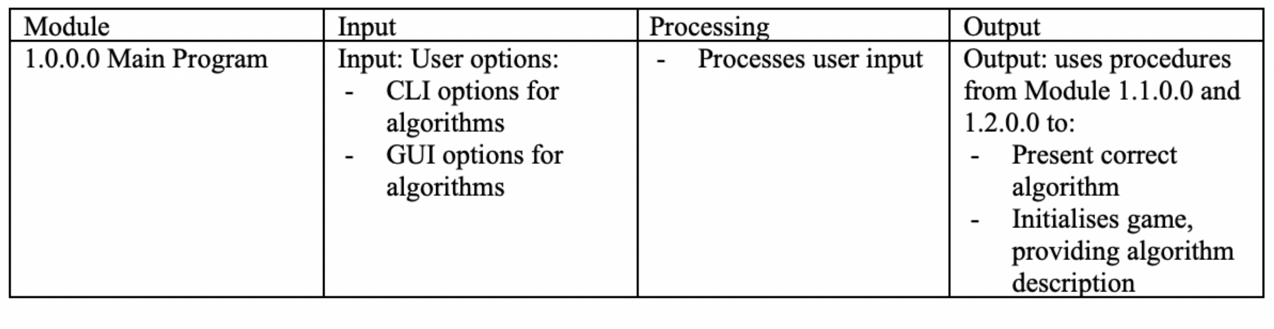
MCTS is favourable over Minimax since it traverses states that are relevant to the current state, whereas Minimax considers every possible move, and as the state space grows, states are merely extra nodes on the tree whereas for a depth-search method such as Minimax, this increases the time taken for evaluation. Despite this, the simulation time will take longer as there would be more states to simulate, so the MCTS algorithm will have to choose the optimal visited node, regardless of whether there is a better node or not. When comparing the algorithms, the MCTS provides a similar solution to the alpha-beta pruning optimisation of the Minimax algorithm through the use of limiting factors, but with a larger space and time complexity. The success of a Monte Carlo Tree Search implementation is based on the average of previous experiences - the inexactitude of the algorithm plateaus over time. Minimax has less inaccuracies, and is more favourable as it doesn't rely on heurisitics. Both algorithms have methods of optimisation, but MCTS typically uses ANNs to not only provide the heuristics for the game simulations, but to speed up the process. As mentioned in Equation 1.3.1, the computational power to conduct these calculations is infeasible with the clients machines. Because of this, the decision-making algorithm I've chosen to implement is the **Minimax** algorithm.

Chapter 2

Design

2.1 Overall Modular Diagram

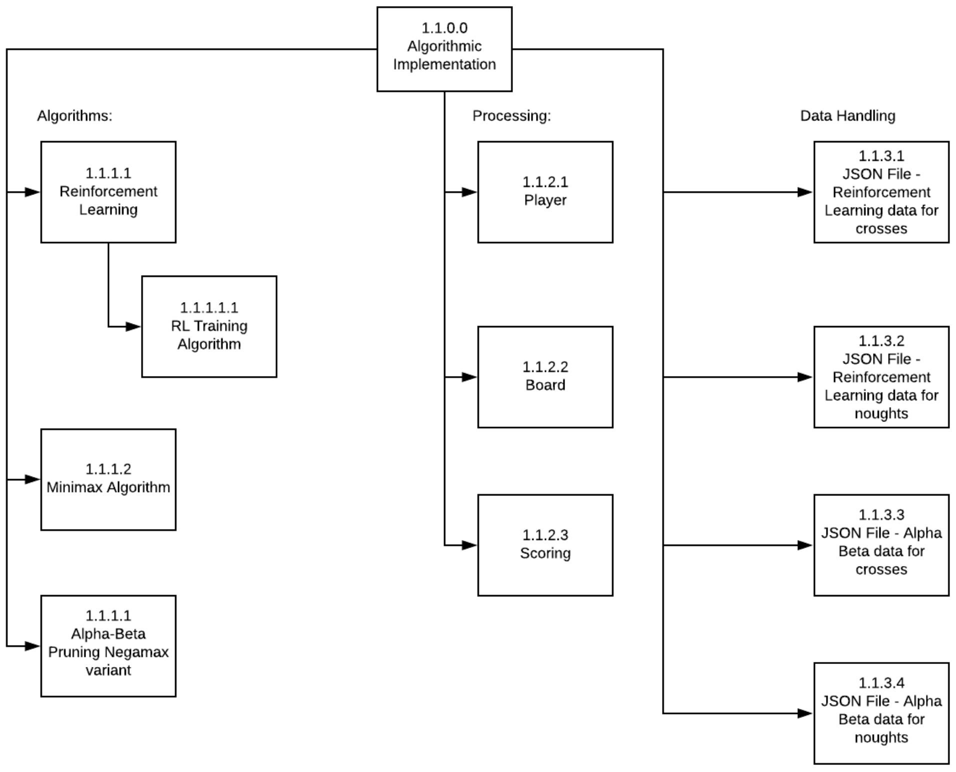




Module	Input	Processing	Output
1.0.0.0 Main Program	Input: User options: - CLI options for algorithms - GUI options for algorithms	- Processes user input	Output: uses procedures from Module 1.1.0.0 and 1.2.0.0 to: - Present correct algorithm - Initialises game, providing algorithm description

Main development is split into two components: the scripting of the chosen decision-making algorithms, and the way the user interacts with those algorithms. To assist my development, I broke the general processes behind the project into several modules, as presented below.

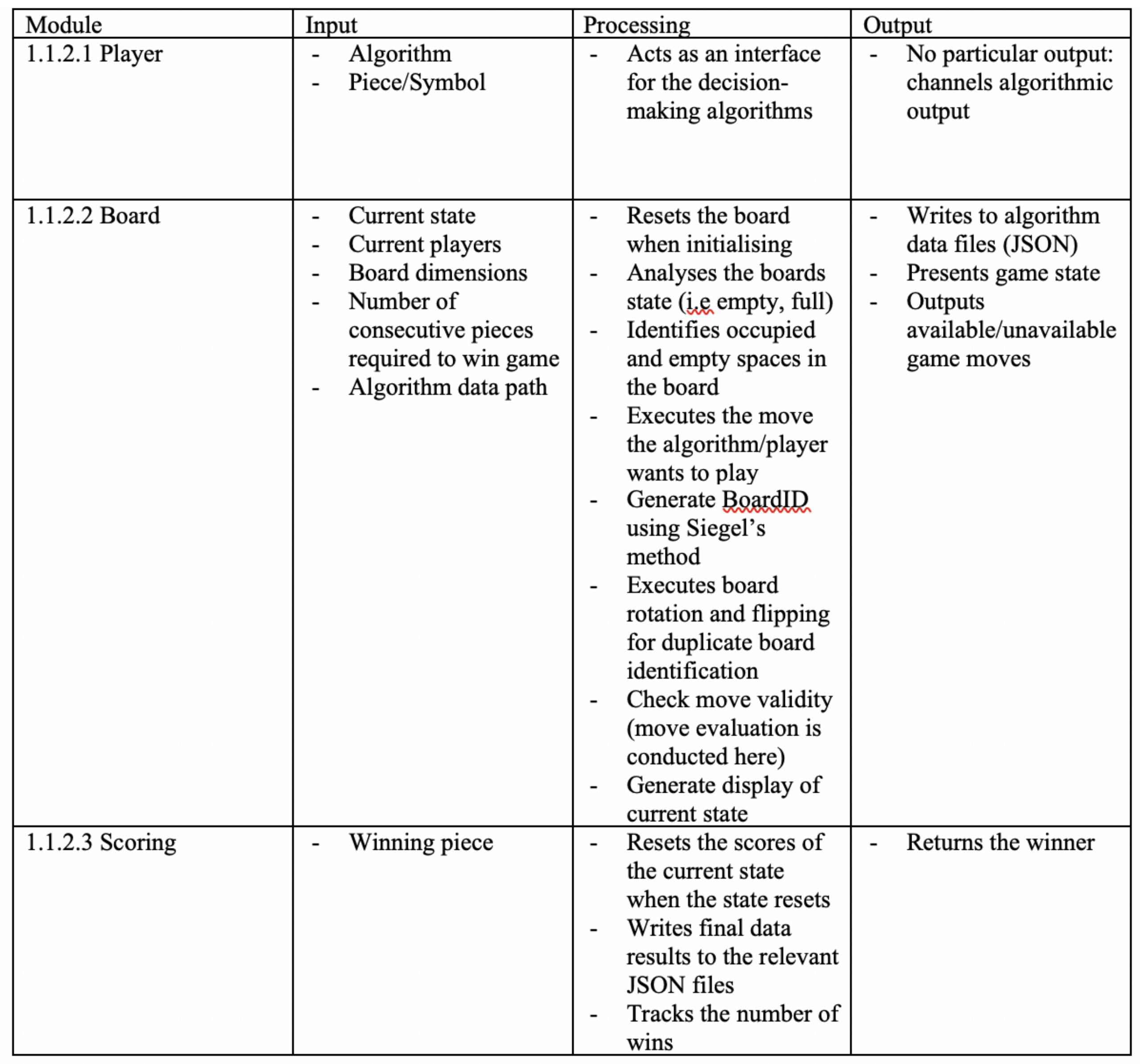
2.1.1 Algorithm-Implementation Module


1.1.0.0 Algorithmic Implementation
Algorithms:
Processing:
Data Handling
1.1.1.1 Reinforcement Learning
1.1.1.1.1 RL Training Algorithm
1.1.1.2 Minimax Algorithm
1.1.1.1 Alpha-Beta Pruning Negamax variant
1.1.2.1 Player
1.1.2.2 Board
1.1.2.3 Scoring
1.1.3.1 JSON File - Reinforcement Learning data for crosses
1.1.3.2 JSON File - Reinforcement Learning data for noughts
1.1.3.3 JSON File - Alpha Beta data for crosses
1.1.3.4 JSON File - Alpha Beta data for noughts

2.2 Player and Board Schematic

I will create a Player class, which any user can instantiate and fill in their relevant details. As for the implementation of the decision-making algorithms for the computer, the class would be cluttered with code irrelevant to the human player. I will create a class which will parse the move from the board, and feed the state into the algorithm's object. This layer of decomposition will help with development, as it separates the algorithms into different segments that I can work on individually. The algorithm object will have a parent class, called "Algorithm." Following the "*Favour composition over inheritance*" rule of thumb, all algorithm variants will inherent this base class, and override inherited functions as well as include algorithm-specific functions. The Board is the state space, so I will program an object that will provide subprocedures that allow for state transitions. State spaces will be provided IDs that are calculated based on its contents, as demonstrated by Siegel's Introductory ML project [**1**].

2.2.1 Player and Board modular decomposition



Module	Input	Processing	Output
1.1.2.1 Player	- Algorithm - Piece/Symbol	- Acts as an interface for the decision-making algorithms	- No particular output: channels algorithmic output
1.1.2.2 Board	- Current state - Current players - Board dimensions - Number of consecutive pieces required to win game - Algorithm data path	- Resets the board when initialising - Analyses the boards state (i.e empty, full) - Identifies occupied and empty spaces in the board - Executes the move the algorithm/player wants to play - Generate BoardID using Siegel's method - Executes board rotation and flipping for duplicate board identification - Check move validity (move evaluation is conducted here) - Generate display of current state	- Writes to algorithm data files (JSON) - Presents game state - Outputs available/unavailable game moves
1.1.2.3 Scoring	- Winning piece	- Resets the scores of the current state when the state resets - Writes final data results to the relevant JSON files - Tracks the number of wins	- Returns the winner

2.3 Algorithmic Implementation

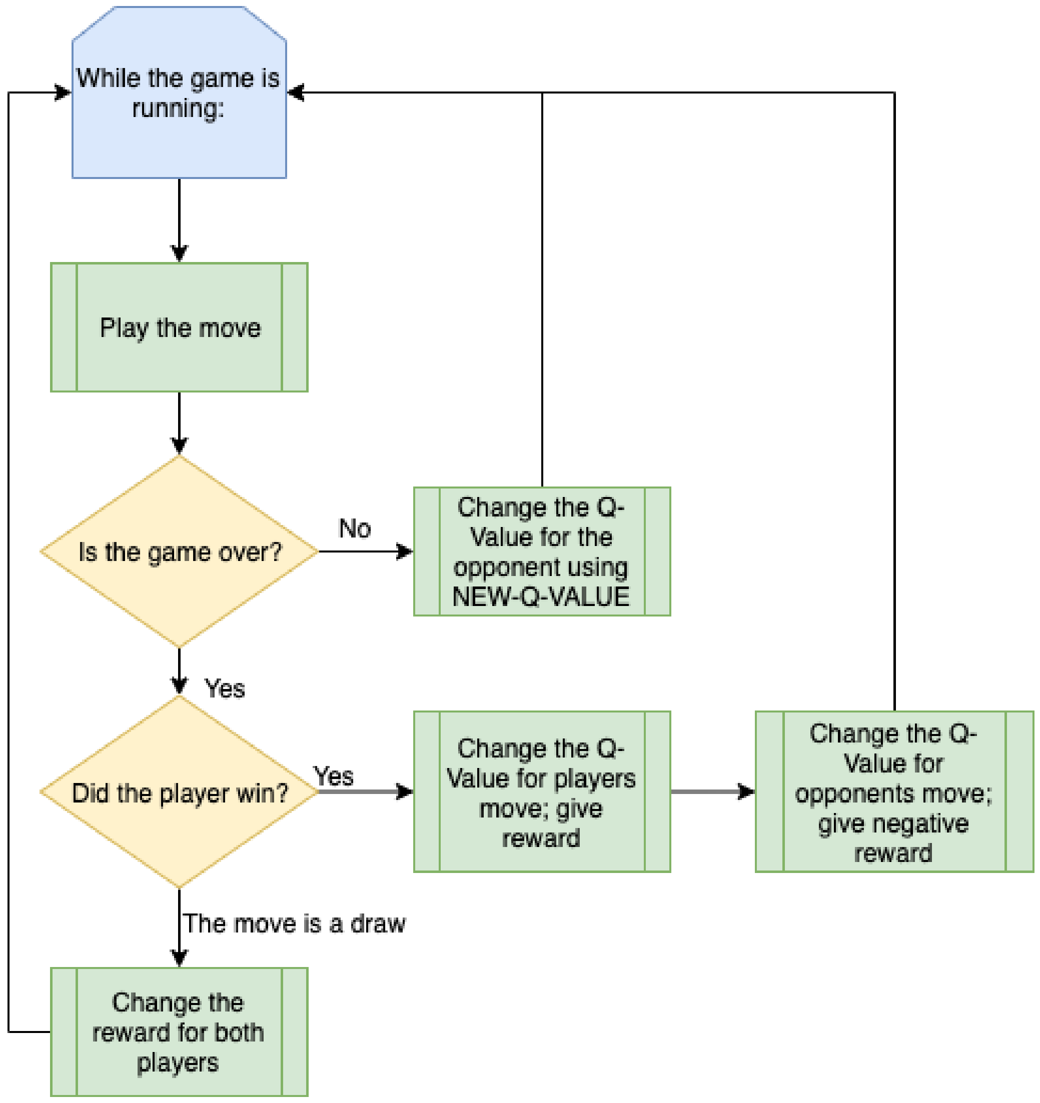
2.3.1 RL Implementation

As mentioned in Section 1.3.1, Reinforcement Learning utilises a Q-Table, ranking future possible moves with a score that the machine can refer to when it's exploiting its current knowledge. We use the Q-Function (1.3.1) to calculate Q-Value, using the previous state, the new state and the reward constant we define. We can use a subprocedure in a class to compute this:

New-Q-Value

```
FUNCTION NEW_Q_VALUE(prevBoard, prevMove, currentBoard, REWAED) ->
    previous_value <- qTable[prevBoard][prevMove]
    ideal_outcome <- maximum value from qTable[currentBoard]
    // The Q-Function in Section 1.2.1
    new_Q_value <- (1 - ALPHA) + ALPHA * (REWARD + GAMMA * ideal_outcome)
    qTable[prevBoard][prevMove] <- new_Q_value
ENDFUNCTION
```

The parameters are dependent on the type of Q-Learning the machine tries, be it exploration or exploitation. But how does the program decide which to go for? We introduce a variable, denoted as Zeta (ζ). Zeta is some random float between 1 and 0, where 1 and 0 represent our two possible options. By using conditional statements, we can compare our Zeta value to the randomly generated value the algorithm produces, and thus it decides whether it exploits or explores. These subprocedures will link back to a main procedure in the class, which will contain a conditional loop. For each iteration, the game will check for a win. In this situation, it will give the winning move a higher Q-ranking score, with the losing players move being given a negative score to prevent the machine from making such a move. If the game results in a draw, then both final moves will be awarded a positive Q-score as they neither won nor lost the game. Until the game reaches a terminal state, the conditional loop will continuously update the Q-score of the moves made as they do not cause a terminal state.


While the game is running:
Play the move
Is the game over?
No
Change the Q-Value for the opponent using NEW-Q-VALUE
Yes
Did the player win?
Yes
Change the Q-Value for players move; give reward
Change the Q-Value for opponents move; give negative reward
The move is a draw
Change the reward for both players

2.3.2 Minimax Implementation

The minimax implementation, paired with alpha-beta pruning, is a little more complicated than the reinforcement learning method. Minimax uses recursion to traverse the game tree, and find the optimal move to play. A game tree can be visualised as follows:

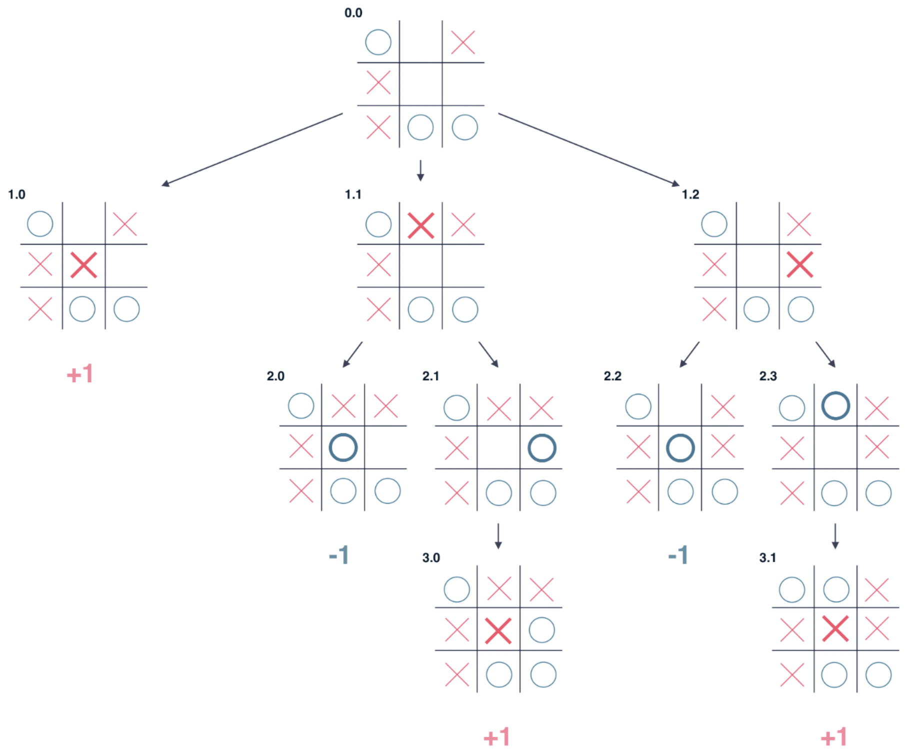


Figure 2.1: *A game tree*

When the algorithm begins, it generates a game tree, weighing each node with a value. The players are the **Minimiser** and the **Maximiser**, with the goal of maximising their benefit and minimising the cost. The algorithm uses a depth-first traversal, relative to the depth-parameter the programmer sets. This means the algorithm will evaluate every state until it reaches a terminal state, and returns the optimal move the player/AI should make. We use the values ∞ and -∞ to signify the detrimental moves - i.e: a move resulting in a win for the opponent. Consider the following game tree snippet:

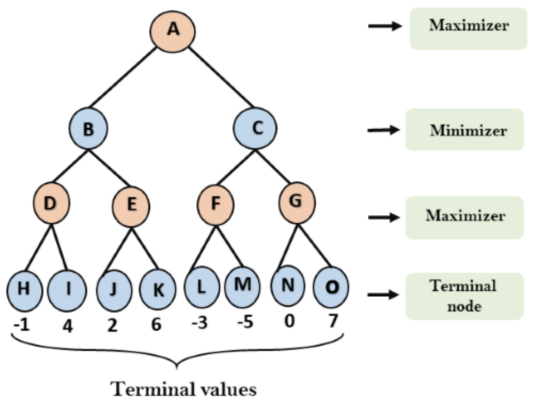


Figure 2.2: *A screenshot of the minimax algorithm at work; the colours signify whose go it is.*

We consider the leaf nodes, and choose the maximum score from it. For the case of Nodes N and O, we would assign node G the value of 7, since 7 is greater than 0. For the minimiser, we take the minimum scores from the nodes and follow this pattern till we reach the root node.

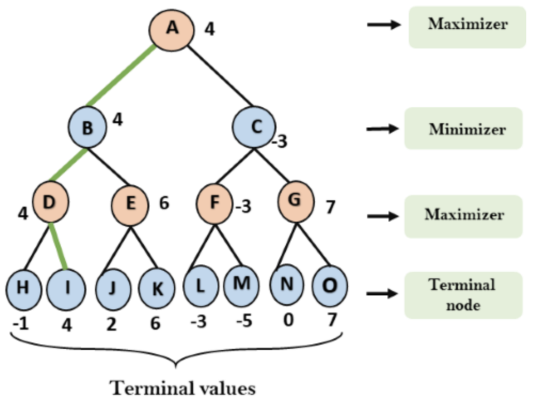


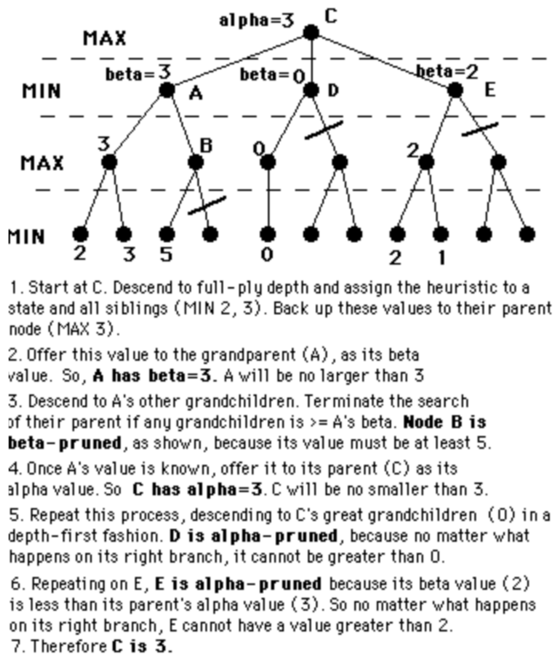
Figure 2.3: *The tree after being traversed, with the relevant scoring assigned to the nodes.*

This is commonly done through iterative processes, but I will use the Negamax variation which uses recursive processes to traverse the trees, the traditional way of parsing data from the tree data structure. The recursive algorithm has a number of steps:

- The base case is to check whether the state is terminal. In the instance it is, return the score back to the main program.
- In the instance it's not the base case, we traverse the tree. When we reach a state, we simulate the move and apply the evaluatory function to see whether the move was a winning or losing move. Once we get a value for this move, the program will either choose to play it or discard it depending on the score given by the function.

- Then we return the best move to the Minimiser or Maximiser

To counteract unnecessary branches being traversed, we use the alpha-beta pruning method. Alpha-beta pruning is an optimisation method for the minimax algorithm, which square roots the exponential time complexity of the algorithm, with $O(N^{\frac{d}{2}})$ (the reader should recall that for some value $n, n^{\frac{1}{2}} \equiv \sqrt{n}$ by laws of indices). Alpha-beta pruning use two parameters, called α (alpha) and β (beta). α records the highest value node, starting at -∞, whereas β records the lowest value node, starting at ∞. **Alpha-beta pruning must fulfill the condition, where $\alpha \geq \beta$.** Only the relevant player can change their respective value; the maximiser affects the α value, and the minimiser affects the β value. Sometimes, the value assigned is an lower or upper bound, so we assign a flag alongside the score telling us, the programmer, whether it's an exact score or a boundary value. The alpha value holds the score of the best possible move, and as we backpropagate through the tree we compare the alpha-beta value to the nodes value to determine which move is better to play. If a nodes score is less than the alpha-beta value, we truncate that branch so the AI no longer considers moves detrimental to its strategy.



1. Start at C. Descend to full-ply depth and assign the heuristic to a state and all siblings (MIN 2, 3). Back up these values to their parent node (MAX 3).

2. Offer this value to the grandparent (A), as its beta value. So, **A has beta=3.** A will be no larger than 3

3. Descend to A's other grandchildren. Terminate the search of their parent if any grandchildren is >= A's beta. **Node B is beta-pruned**, as shown, because its value must be at least 5.

4. Once A's value is known, offer it to its parent (C) as its alpha value. So **C has alpha=3.** C will be no smaller than 3.

5. Repeat this process, descending to C's great grandchildren (0) in a depth-first fashion. **D is alpha-pruned**, because no matter what happens on its right branch, it cannot be greater than 0.

6. Repeating on E, **E is alpha-pruned** because its beta value (2) is less than its parent's alpha value (3). So no matter what happens on its right branch, E cannot have a value greater than 2.

7. Therefore **C is 3.**

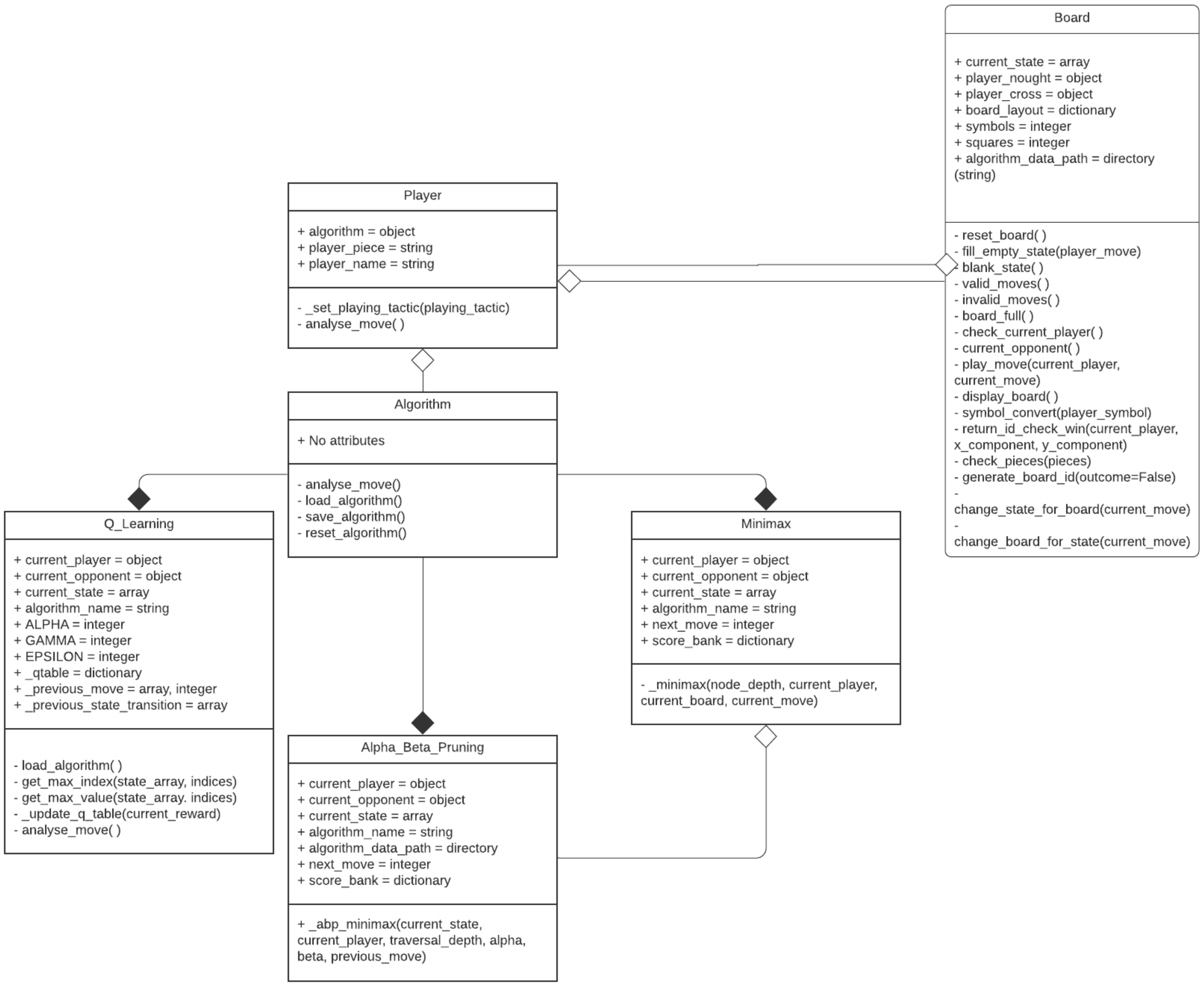
Figure 2.4: *An example of alpha-beta pruning from a first year undergraduate textbook.* [**2**]

But what if the algorithm encounters a board that it analysed earlier in the game? By using a dictionary, I plan on implementing a reference table so the algorithm checks whether it's a visited node. If it isn't, it will explore the node but if it is, it will use the previously allocated score rather than recalculating it.

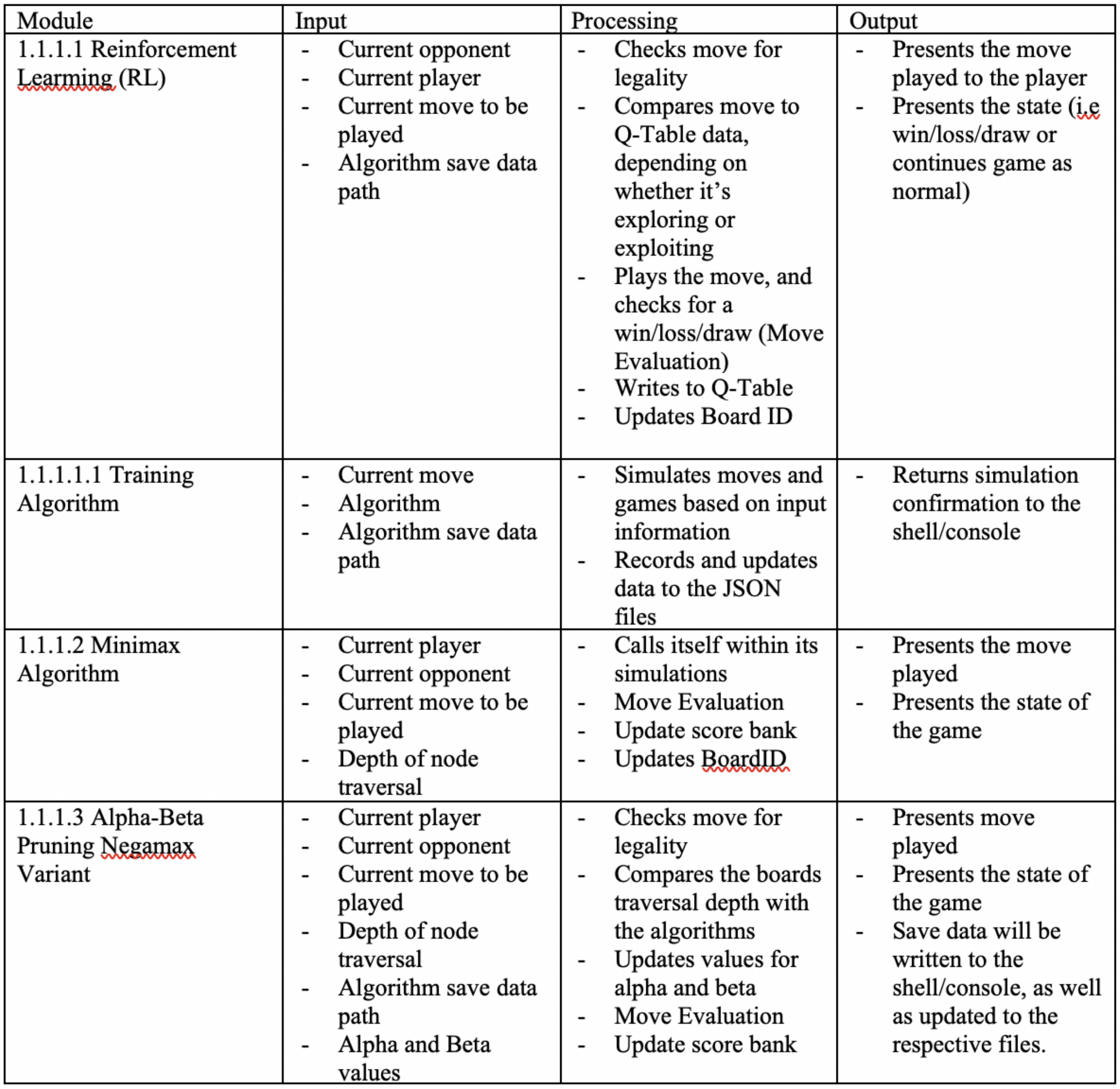
Negamax Implementation Pseudocode

```
FUNCTION Negamax(alpha, beta, current_board, player) ->
    alpha_0 <- alpha
    // We record the original alpha value so we can use it as reference
    // for which flag to use

    score_table = {}
    // We'll track the scores of the states in a dictionary that
    // we can reference later

    optimal_move <- (score <-  -infinity, move <- -1)
    // I could implement a tuple for the score and move procedure

    IF current_board.state IS terminal ->
        IF player.status IS "win" ->
            optimal_move[ score ] <- 1
        ELSE IF player.status IS "lose" ->
            optimal_move[ score ] <- -1
        ELSE
            optimal_move[ score ] <- 0

    return optimal_move
    ENDIF

    IF current_board IN reference table ->
        IF the depth of the entry > current_board.depth ->
            IF current_state[flag] IS "lower" ->
                alpha <- MAX( alpha, current_state[score] )
            ELSE IF current_state[flag] IS "upper" ->
                beta <- MIN( beta, current_state[score] )
            ELSE IF current_state[flag] IS "exact score" ->
                optimal_move[score] <- current_state[score]
                optimal_move[move] <- current_state[move]
            ENDIF

            IF alpha is greater than beta ->
                optimal_move[move] <- current_state[move]
            ENDIF
ENDIF

    FOR move IN every possible move ->
        simulate the move
        score_table[move] <- Negamax(board, opponent)[score]
        undo the move
        alpha <- MAX(alpha, score_table[move])
        IF alpha is greater than beta ->
             BREAKLOOP
        ENDIF
    ENDFOR

    FOR move and score in score_table ->
        IF score is greater than optimal_move[score] ->
            optimal_move <- ( score_table[move], score_table[score] )
        ENDIF
    ENDFOR

    CALCULATE_FLAG(alpha_0, beta, optimal_move)
    store the board in the reference table
    return optimal_move

ENDFUNCTION

FUNCTION CALCULATE_FLAG(alpha, beta, move) ->
    FLAG <- " "
    IF move[score] is greater than or equal to beta ->
        FLAG <- "lower"
    ELSE IF move[score] is less than or equal to alpha ->
        FLAG <- "upper"
    ELSE
        FLAG <- "exact score"
    return FLAG
ENDFUNCTION
```

2.3.3 Initial UML Diagram


Board
+ current_state = array
+ player_nought = object
+ player_cross = object
+ board_layout = dictionary
+ symbols = integer
+ squares = integer
+ algorithm_data_path = directory (string)
- reset_board( )
- fill_empty_state(player_move)
- blank_state( )
- valid_moves( )
- invalid_moves( )
- board_full( )
- check_current_player( )
- current_opponent( )
- play_move(current_player, current_move)
- display_board( )
- symbol_convert(player_symbol)
- return_id_check_win(current_player, x_component, y_component)
- check_pieces(pieces)
- generate_board_id(outcome=False)
- change_state_for_board(current_move)
- change_board_for_state(current_move)
Player
+ algorithm = object
+ player_piece = string
+ player_name = string
- _set_playing_tactic(playing_tactic)
- analyse_move( )
Algorithm
+ No attributes
- analyse_move()
- load_algorithm()
- save_algorithm()
- reset_algorithm()
Q_Learning
+ current_player = object
+ current_opponent = object
+ current_state = array
+ algorithm_name = string
+ ALPHA = integer
+ GAMMA = integer
+ EPSILON = integer
+ _qtable = dictionary
+ _previous_move = array, integer
+ _previous_state_transition = array
- load_algorithm( )
- get_max_index(state_array, indices)
- get_max_value(state_array. indices)
- _update_q_table(current_reward)
- analyse_move( )
Minimax
+ current_player = object
+ current_opponent = object
+ current_state = array
+ algorithm_name = string
+ next_move = integer
+ score_bank = dictionary
- _minimax(node_depth, current_player, current_board, current_move)
Alpha_Beta_Pruning
+ current_player = object
+ current_opponent = object
+ current_state = array
+ algorithm_name = string
+ algorithm_data_path = directory
+ next_move = integer
+ score_bank = dictionary
+ _abp_minimax(current_state, current_player, traversal_depth, alpha, beta, previous_move)

2.3.4 Algorithm modular decomposition



Module	Input	Processing	Output
1.1.1.1 Reinforcement Learming (RL)	- Current opponent - Current player - Current move to be played - Algorithm save data path	- Checks move for legality - Compares move to Q-Table data, depending on whether it's exploring or exploiting - Plays the move, and checks for a win/loss/draw (Move Evaluation) - Writes to Q-Table - Updates Board ID	- Presents the move played to the player - Presents the state (i.e win/loss/draw or continues game as normal)
1.1.1.1.1 Training Algorithm	- Current move - Algorithm - Algorithm save data path	- Simulates moves and games based on input information - Records and updates data to the JSON files	- Returns simulation confirmation to the shell/console
1.1.1.2 Minimax Algorithm	- Current player - Current opponent - Current move to be played - Depth of node traversal	- Calls itself within its simulations - Move Evaluation - Update score bank - Updates BoardID	- Presents the move played - Presents the state of the game
1.1.1.3 Alpha-Beta Pruning Negamax Variant	- Current player - Current opponent - Current move to be played - Depth of node traversal - Algorithm save data path - Alpha and Beta values	- Checks move for legality - Compares the boards traversal depth with the algorithms - Updates values for alpha and beta - Move Evaluation - Update score bank	- Presents move played - Presents the state of the game - Save data will be written to the shell/console, as well as updated to the respective files.