EXPLORATION OF AI EFFECTIVENESS WITHIN THE GAME OF CASTLE

Student: Cyrus Dobbs
Supervisor: Dr Frank C Langbein
Moderator: David J Humphreys

One Semester Individual Project (CM3203)

40 Credits

TABLE OF CONTENTS

Figures

Tables

1 Introduction

This chapter will focus on the project's objectives, intended audience and outcomes. The author will also lay out the fundamentals of their approach to the problem and briefly introduce the game of Castle.

1.1 Project Aim and Scope

The aim of this project is to experiment with different AI agents that compete at the game of Castle. The end goal is to have produced an AI agent that can compete with a human player. While the scope of the overall problem could be a complete, marketable game, the author instead chooses to concentrate their limited time available primarily on the AI aspect of the game. With time being one of the biggest constraints on the project, producing online capabilities and a graphical user interface (GUI) was omitted from the project's scope. However, a working game engine with limited terminal-based UI is still necessary in order to achieve the project's goal of producing a challenging AI agent.

The game selected for the project is called Castle[1], a 2-4 player card game. This game was chosen because in relation to other card games, Castle is not widely known and is, therefore, lacking in prior work. From the context of functioning as an AI testbed, Castle provides interesting learning opportunities due to its properties of imperfect information, as players hide their cards from opponents, and randomness, caused by the shuffled deck of cards. This makes it more difficult for the AI agent to determine a set of moves that will take it closer to winning. Between humans, it is often played at a fast pace with each game not lasting much longer than five minutes. This is helpful because it means many simulations between AI players can be run in a short time frame.

1.2 Intended Audience

One potential audience could be students studying an AI course or perhaps taking on a similar project. Equipped with this report and the source code they would be able to learn:

- How to implement an AI algorithm in the context of a game
- Basic information regarding the machine learning libraries needed to implement a similar system
- Method for simulating and collecting data

Identifying potential student or research beneficiaries highlighted the need to hold the source code in a public repository. This became particularly apparent to the author as during development it was difficult to find similar projects for inspiration and guidance.

Researchers may also have an interest in this piece of work if their work overlaps with this domain. Additionally, researchers may wish to further this project by adding to it themselves. This further highlights the need for the authors' work to be lodged in an accessible and public space.

[1] https://en.wikipedia.org/wiki/Castle_(card_game)

It is the author's opinion that a game developer interested in implementing an AI player may find it useful to learn from this project's AI players.

1.3 Approach Chosen to Solve the Problem

Firstly, a game engine must be produced that can simulate a game of Castle. This engine will be made in a way that makes player types and their strategies modular. By using this approach throughout development of different AI players, the engine will be able to be configured to play with different players with ease.

Secondly, an Information Set Monte Carlo Tree Search (ISMCTS) algorithm will be used to produce an AI agent that can play the game from only a set of available moves and a game end condition. The agent will have no game specific knowledge. Additionally, a rule-based agent with a set strategy will also be produced. Thousands of simulations of games between these two 'players' will then be run in order to collect a database of game states. These game states are snapshots of the state of the game at the beginning of each turn for the ISMCTS player, including who won the game that the game state was taken from.

Using this database of game states, the author aims to experiment with various Machine Learning (ML) libraries in order to produce a model for evaluating a given state. This can then be used by the AI agents in order to get a value for how promising a state is.

1.4 Important Outcomes

A basic rule-based AI will be used as a benchmark for the different AI agents produced. The win-rate of the various versions of AI against this benchmark will be used to evaluate the success of chosen techniques. The most important outcome of this project is to have produced a successful AI agent. The level of success observed from the benchmark tests will serve as the evaluation of this success. Additionally, the success of the machine learning aspect of the project will also be an important outcome, measured by the accuracy rate, among other statistics. If the AI agents are not successful, it is important for them to be at a stage of development that can easily inspire and support future work.

2 Background

Following on from the introduction, the background will give the reader more in-depth knowledge of concepts fundamental to the project. The rules of Castle will be outlined before describing the chosen algorithm, ISMCTS, used as the AI player's basic strategy. ISMCTS will be briefly contrasted to other techniques before providing some relevant existing solutions to similar problems.

2.1 Wider Project Context

In 1997, Deep Blue [1] beat the World Chess Champion Garry Kasparov in a landmark victory for artificial intelligence. Deep Blue used minimax with alpha beta pruning and a heuristic evaluation function in order to pick its moves. More recently, Google's DeepMind[2] has produced AlphaGo and AlphaStar which both compete at the top match levels of Go and StarCraft. While Chess has a state-space complexity of 10^{47}, the Chinese game of Go is larger at 10^{170} and StarCraft is even more complex still and has hidden information. This increase in game size and complexity has meant we have had to move away from more traditional searching methods. In a very different approach to Deep Blue, Google's new AIs utilise reinforcement learning.

These advances in AI agents have great and far reaching applications to more practical problems such as weather prediction and climate modelling. Many sectors are utilising AI in order to automate processes while pushing for these systems to make better decisions than humans. New AI systems are likely to dominate industry and aspects of society in the near future and will bring with them new ethical challenges for us to overcome [9].

Video games are extremely useful as test beds and benchmarks for new AI. Togelius explains how video games can test many aspects of intelligence and how in contrast to using robots, they can be sped up in order to run thousands of simulations in very short time spans [2].

2.2 Castle Rules

Castle is a card game played between 2-4 people. The game starts by each player being dealt 3 cards; we will call these the face-down Castle cards (FDCCs). They are to be placed face down on the table remaining hidden to the players until the instance they are played. Next, each player is dealt 6 cards to their hand, hidden from the opponents. The players must then select 3 cards to be placed face up on top of their FDCCs; we will call these the face-up Castle cards (FUCCs).

Players take it in turns to play a card of either the same or higher face value, from their hand, than the last card played (suit has no bearing on the game). Once a card from their hand is played, if that brings the number of cards in their hand to less than 3, they must then pick up a card from the rest of the deck and add it to their hand. This way players always have 3 cards in their hand until the deck has been completely depleted. If a player cannot play a card because the value of the last played card is higher than any cards in their hand, they must pick up the pile of discarded cards and add them to their hand. An

[2] https://deepmind.com/

exception to this rule is if the player has a magic card in their hand, as these cards can be placed on any card (regardless of their face value being lower than the last card played).

There are two types of magic cards:

- All the 2's reset the played pile to this value (the lowest possible), allowing the play to continue.
- All the 10's burn the played pile. This means they are removed from the game and the player then gets another turn to play any card in their hand.

Once the deck has been completely depleted, players can then have less than 3 cards in their hand. Once a player has run out of all cards in their hand, they can then play one of their FUCCs. The fact that the other players can see these FUCCs makes it important for players to put the highest cards from their starting hand down as their FUCCs as this helps to prevent opponents them from making them pick up the pile by playing higher cards. Once a player has no cards in their hand, and no FUCCs, they can then choose a FDCC and try to play it. If they cannot, they must pick up the pile. Play continues until a player wins by having no cards in their hand, FUCCs or FDCCs.

Additional rules:

- A player can play multiple cards at once if they are of the same face value and either all from their hand or all FUCCs.
- If a player plays the 4th card of the same face value onto the pile and they have been played consecutively without any other cards of a different face value in between then the pile is 'burned' and these cards are no longer in the game - in the same fashion as when a 10 is played – and the player may take another turn.

2.3 Algorithm Choices

2.3.1 Information Set Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that focuses on sampling the search space of the game using random moves to simulate many full games. The results of the simulations are backpropagated up the tree in order to guide expansion of new nodes.

Information Set Monte Carlo Tree Search (ISMCTS) [5] is a modification to MCTS[3] that utilises Information Sets (IS). An Information Set is the current information available to the player together with the set of possible permutations of hidden cards (opponents' cards, the deck, etc).

ISMCTS uses the current information set to limit the combinations of determinisations to the possible states of the game – it then uses a single tree to run an iteration of MCTS for each determinisation up to a limit. This limit can be a given number of iterations or a time limit. A decision is then based on the proportion of times a move is traversed over all the simulations.

[3] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

ISMCTS is an 'any time' algorithm, meaning it can be for a given length of time and then takes the best available move at that time. However, there are some potential issues with ISMCTS. For example, classic ISMCTS does not utilise any game specific knowledge and this could lead to some AI moves looking unrealistic and illogical to human players. However, performance is largely proportional to how long the algorithm is run for.

2.3.2 UCB

MCTS encounters the exploration-exploitation trade-off within its 'expansion' step. This arises when needing to maintain balance between exploitation of successful known moves and exploration of unknown moves. Kocsis and Szepesvari introduced a now commonly used algorithm that handles this called the Upper Confidence Bound (UCB) algorithm [3]. Kocsis and Szepesvari recommend picking the move that returns the highest value from the formula seen in Figure 1 (quoted from Wikipedia) [7].

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

FIGURE 1: UCB ALGORITHM

- w_i stands for the number of wins for the node considered after the *i*-th move
- n_i stands for the number of simulations for the node considered after the *i*-th move
- N_i stands for the total number of simulations after the *i*-th move run by the parent node of the one considered
- *c* is the exploration parameter—theoretically equal to √2; in practice usually chosen empirically

2.3.3 Comparison to other algorithms

Another traditional algorithm used is minimax search with alpha beta pruning. However, this presents challenges when applied to the game of Castle. One reason for this is that the game tree for Castle is very large and this would present performance issues, likely making this approach unfeasible. Additionally, minimax relies on heuristics in order to score states or moves within the game. Unlike the game of chess, where it is fairly intuitive to produce heuristics at least at a basic level; for Castle this is much more difficult and likely to be less effective because there is no clear way to rank how a move will increase a player's chance of winning. Classic MCTS removes the need for an evaluation function by always simulating the game to a terminal state.

2.4 Relevant Existing Solutions

2.4.1 Game state evaluation in *Hearthstone*

Jakubik [4] used neural networks in order to evaluate game states from the game of Hearthstone. They collected a training set by playing pairs of MCTS players against each other and recording the results at each turn, along with logging who won the game. They then used these results to train a neural network as an evaluation function using the Theano Python library. The neural network was then evaluated using an AUC score. Both Jakubik

and the author attempt to use neural networks to evaluate game states. However, in contrast to the authors work, Jakubik does not implement the evaluation function within an AI player and therefore provides no gameplay evaluation of its success.

2.4.2 MCTS in *Magic: The Gathering*

Cowling et al explored the use of ISMCTS in the game of Magic: The Gathering (MTG) [7]. They also choose to focus on enhancing the MCTS algorithm, however, they do this without machine learning techniques. One method they explore is adding heuristics in order to evaluate in more depth than the core MCTS method of back-propagation of simulated wins allows. An example of this is the addition of 'Discounted Reward' which incentivises the AI to choose moves that win the game earlier because this prevents the opponent from being able to draw a 'lucky card' that can turn the game in their favour. They achieve this by backpropagating values from custom functions rather than just a binary win/loss.

3 Design and Implementation of Systems

The following section will give an overview of the systems built to support the development of the AI player.

3.1 Castle Game Engine

Firstly, the author needed to produce an application in order to run the game of Castle. Being the sole user, the author needed to be able to configure the application to run from a set of inputs: The type of players in the game, including which strategies they would employ, and an end condition (number of games or a time limit). Executing thousands of simulations, the system needed to be efficient, as any small increase in running time for a single game would be noticeable in the overall running time. Another requirement was that the author would need the flexibility to run the application from the command line of any OS (Windows, Mac, Linux). This software needed to be modular enough that they could add different AI agents as players. It also needed to have a core game state that would hold all the information about the game. For these reasons, the author opted to follow the Model-View-Controller[4] design pattern. This approach demands the separation of concerns, in turn producing low coupling and high cohesion, ultimately making it easier to change and reconfigure as the project progressed. Java was chosen primarily because it was the programming language the author has the most experience working with. However, it is also suitable because it matches the requirements of being multiplatform and efficient.

The next 4 sections will give overviews of each part of the application with the support of UML diagrams. These diagrams are high resolution and may require the reader to zoom.

[4] https://en.wikipedia.org/wiki/Model-view-controller

3.1.1 Game representation

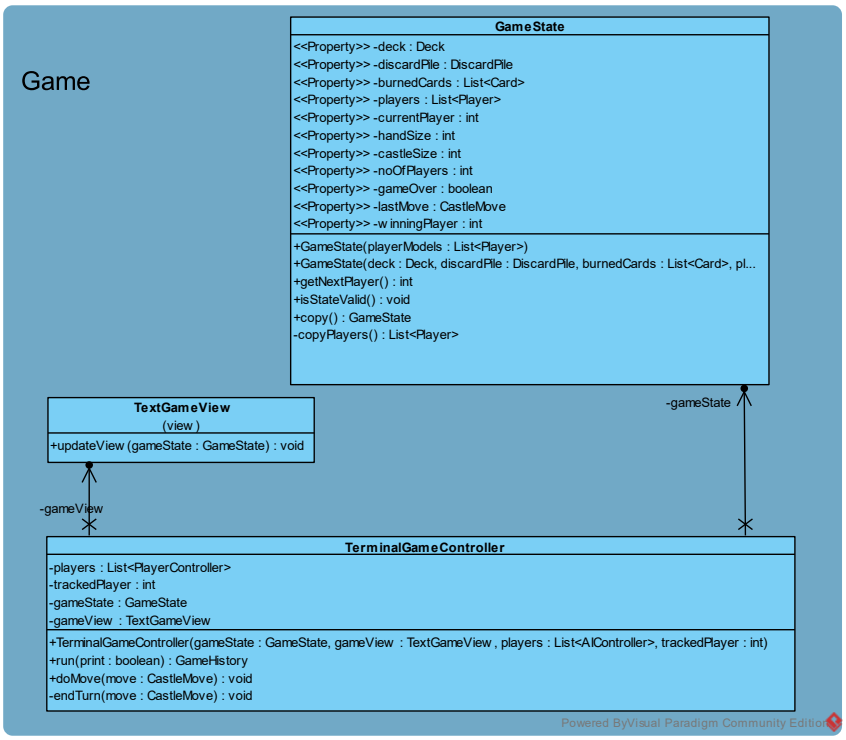

FIGURE 2: GAME REPRESENTATION UML

The game's state is a model within the MVC pattern and is represented by the *GameState* object. The *GameState* holds the information about the game such as the *Deck* (made up of *Card*s), the *Discard Pile*, the *Player*s etc. The logic that alters the state of the game has been extracted to the *TerminalGameController*. The *TerminalGameController* executes the main loop of the game – cycling through players' turns, requesting a *Move* from them, and then executing the move on the *GameState* – until a player has won. Extraction of the logic from the *GameState* allows for future work to add different *GameControllers*, should that ever be a requirement. The *TerminalGameController*, contains the *GameState* as well as the *GameView*; responsible for allowing observation of the game. Currently the only implementation of GameView is the *TextGameView* that renders the game to the terminal. However, the system allows for a different *GameView* in the future – for example, a different version of the current terminal view or even a GUI.

3.1.2 Player representation

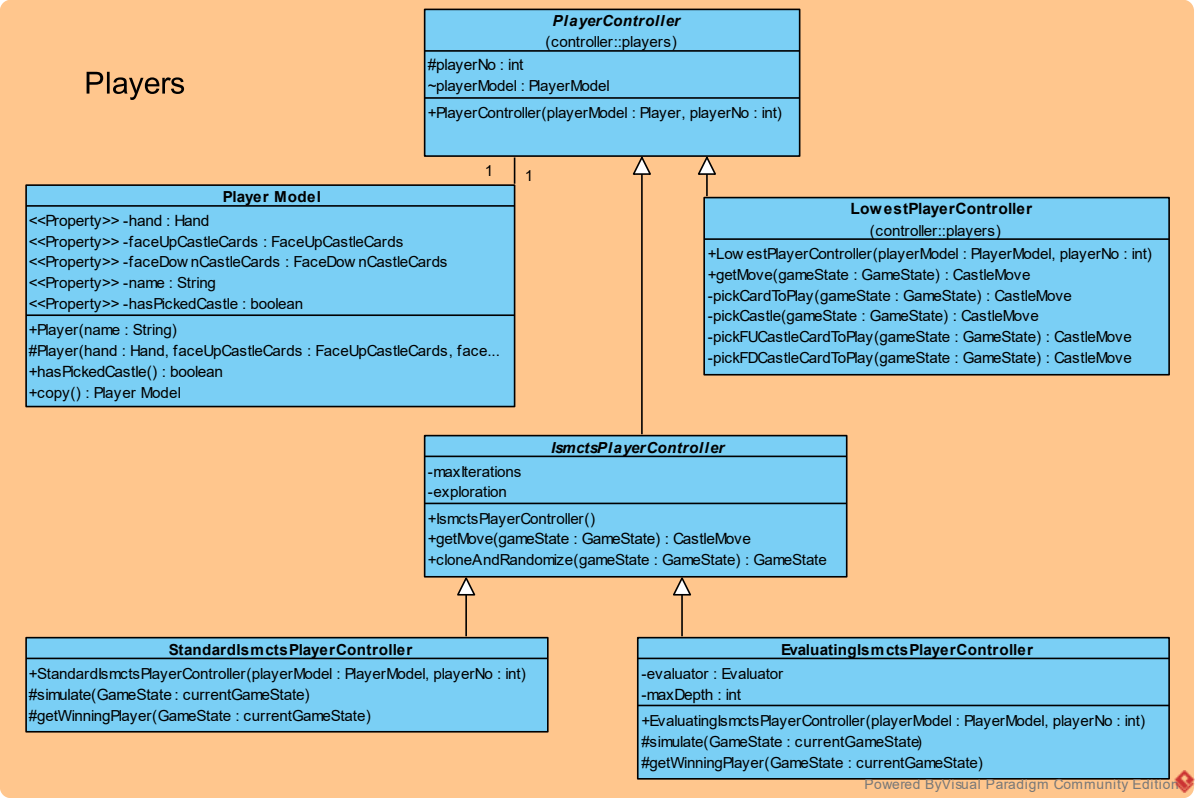

FIGURE 3: PLAYER REPRESENTATION UML

Like the game, the player representation consists of a *PlayerController* that contains the player number and the *PlayerModel. PlayerController* is an abstract class that is extended by each type of player. For example, the *LowestPlayerController* is the rule-based AI strategy that selects the move with the lowest card. This selection logic is within the *LowestPlayerController* where it then passes the selected move to the *GameController*. Where the AI controllers select the cards, the *HumanPlayerController* offers the potential moves to the player through the terminal and then passes the player's selection to the *GameController* for it to be executed, and then for the *GameState* to be changed accordingly.

3.1.3 Moves

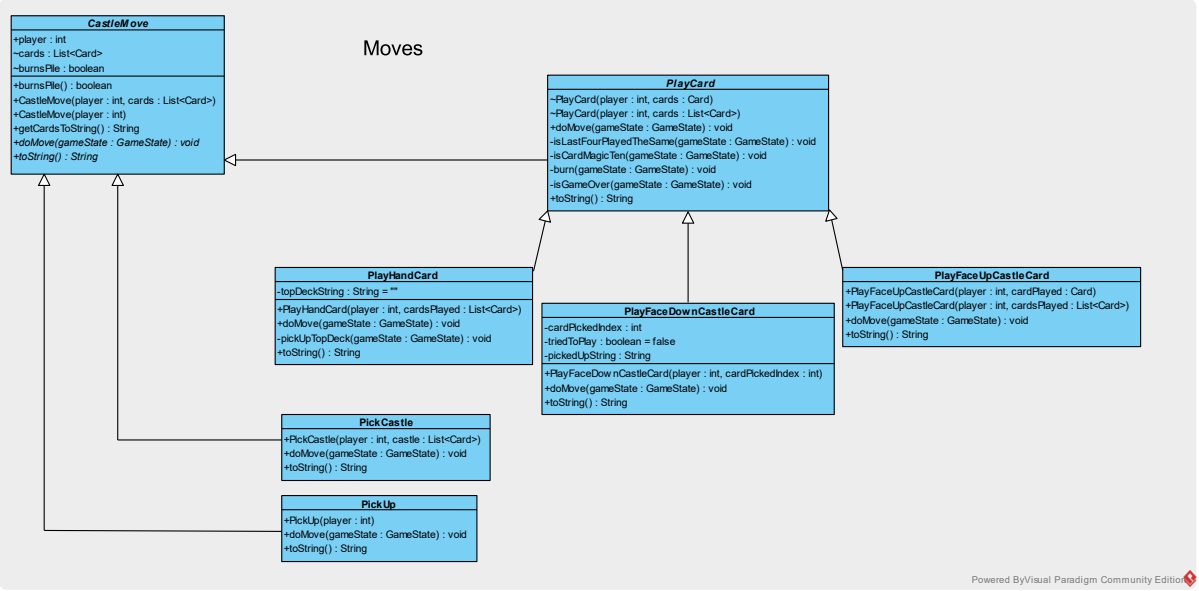

FIGURE 4: CASTLE MOVES UML

Moves are the actions within the game that players can make. *CastleMove* is an abstract class and serves as the interface by which the *GameController* makes use of the moves. For example, the *GameController* only knows of the generic *doMove* method that is implemented differently for each type of move. When the GameController calls *doMove*, it passes the *GameState* to the move object, where it carries out the logic of the move on the state of the game.

There are 2 types of abstract moves and 5 concrete moves:

- (Abstract) **CastleMove** - This is the highest-level move in the architecture and acts as the interface to all the other moves with its abstract *doMove* method. The next 6 moves implement *doMove*.
    - **PickCastle** – This move removes the 3 chosen cards from the players hand and adds them to their FUCCs
    - **PickUp** – This is used if the player cannot do any other move and puts all the discarded cards into the players hand
    - (Abstract) **PlayCard** – This is another abstract move with some implementation of *doMove*. However, the next 3 moves add further implementation while extending PlayCard.
        - **PlayFaceDownCastleCard** - This move removes the chosen card(s) from the players FDCCs and adds them to the discard pile
        - **PlayFaceUpCastleCard** - This move removes the chosen card(s) from the players FUCCs and adds them to the discard pile
        - **PlayHandCard** - This move removes the chosen card(s) from the players Hand and adds them to the discard pile

3.1.4 Setup and Run

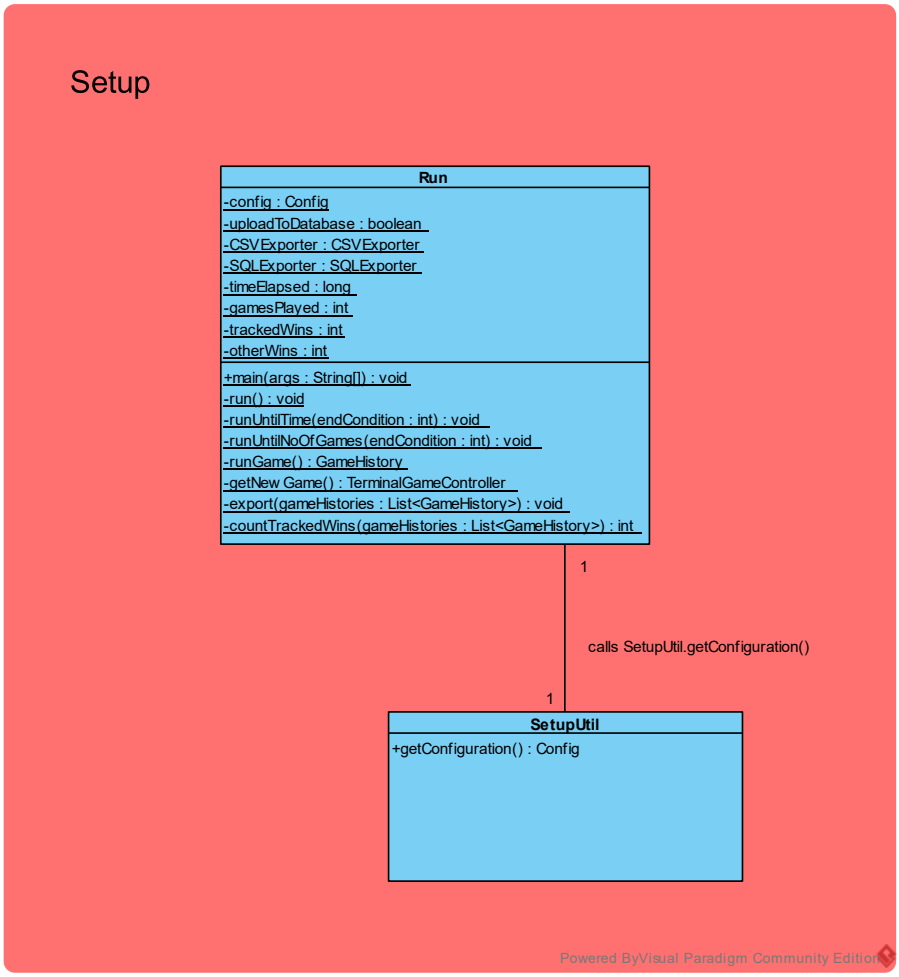

FIGURE 5: SETUP UML

Once the user executes the jar file from the command line, they are presented with a series of options in order to set up the application in the desired configuration. The following options are presented to the user:

1. The type of end condition:
    1) No. of games
    2) Time limit
    3) Indefinite
2. The integer end condition value:
    1) No. of games
    2) Time in minutes
3. Print the games in real-time to the console:
    1) True
    2) False
4. Export the simulation results to CSV:
    1) True
    2) False
5. Export the results to database:
    1) True
    2) False
6. Number of players:
    1) 2
    2) 3
    3) 4
7. For each player, the type of player must be selected. Some players require additional configuration.
    1) Human
    2) ISMCTS
    3) Customer Evaluation ISMCTS
    4) Lowest-card strategy
    5) Random decisions
8. Alternate which player goes first?
    1) True
    2) False

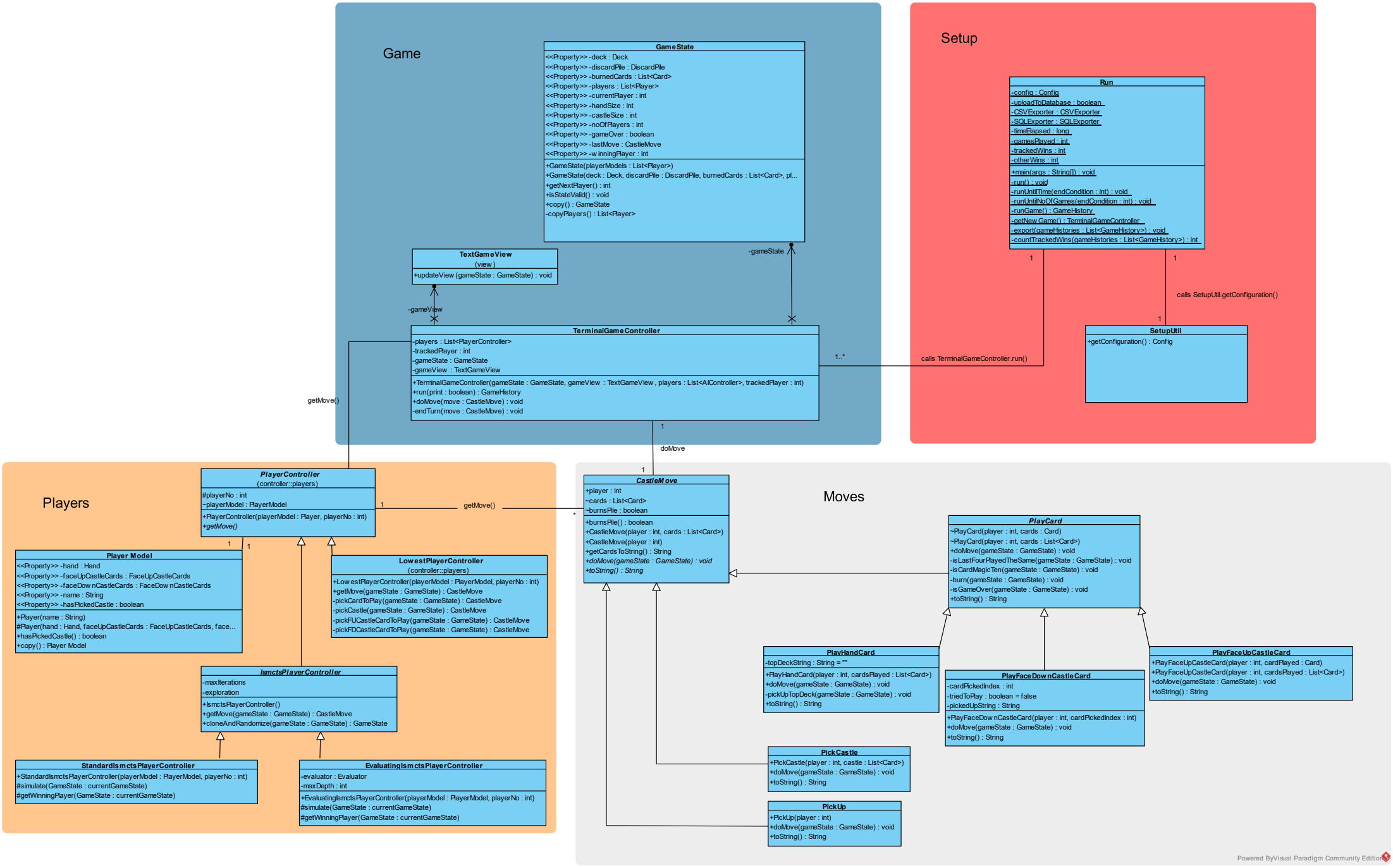

FIGURE 6: CASTLE ENGINE UML

3.2 ISMCTS Implementation

In this section, the author will expand on the high-level explanation of ISMCTS that was presented in section 2.3.1 and then walk through the coded implementation used for this project.

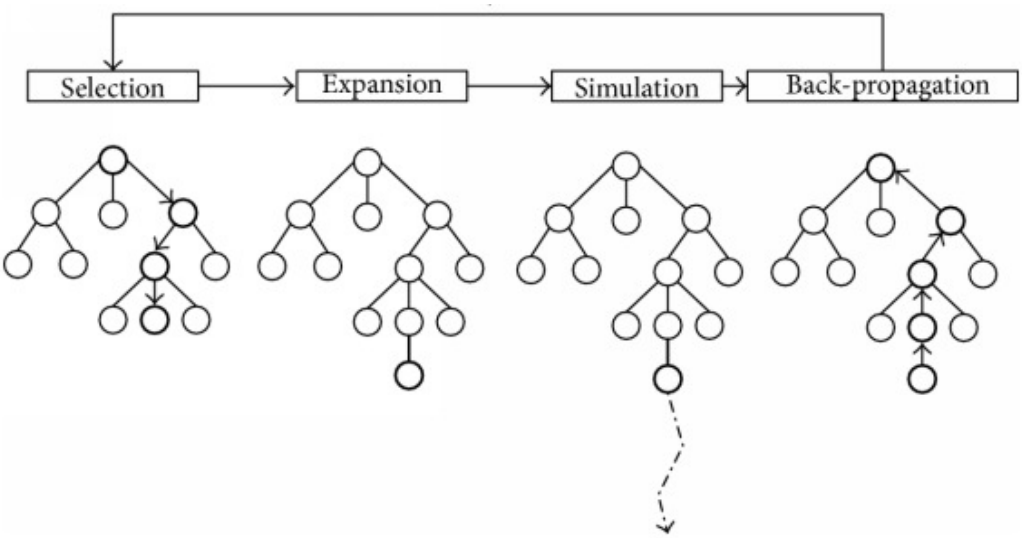

FIGURE 7: MCTS STAGES

Figure 7 illustrates the four main stages to the MCTS algorithm. These steps are repeated for a given number of iterations or until some other end condition is met (e.g. a time limit).

> **Selection**: Move down the tree by traversing nodes selected by the UCB algorithm until reaching a leaf node.
>
> **Expansion**: Add a node to the tree corresponding to an available untried move, selected at random.
>
> **Simulation**: Run a simulation of a game from the expanded node by selecting random moves until reaching a terminal state.
>
> **Back-propagation**: Carry the result of the simulation back up the tree to update the count of wins in the expanded and selected nodes for that iteration.

We used an end condition (`maxIterations`) of 3200 iterations consistent throughout development and evaluation in order to ensure all test results were comparable.

```
public CastleMove getMove(GameState gameState) {

    Node rootNode = new Node();
    for (int i = 0; i < maxIterations; i++) {
        Node node = rootNode;

        // Determinisation
        GameState currentGameState = cloneAndRandomize(gameState);
```

A determinised game state is produced from the current state; all the locations of cards unobservable to the player are randomised. This part of the algorithm allows us to take a

random sample of game states from the set of possible game states that the player could be in.

Next is the Selection stage. Starting at the root, we select child nodes via the UCB algorithm (see UCB) until we reach a node with untried moves.

```
// Selection
List<CastleMove> availableMovesForSelection = getMoves(currentGameState);
while (!availableMovesForSelection.isEmpty()
        && node.getUntriedMoves(availableMovesForSelection).isEmpty()) {

    node = node.UCBSelectChild(availableMovesForSelection, exploration);
    doMove(node.getMove(), currentGameState);
}
```

In the next stage, Expansion, we add a node to the tree corresponding to a randomly selected available move.

```
// Expansion
List<CastleMove> availableMovesForExpansion = getMoves(currentGameState);
if (!node.getUntriedMoves(availableMovesForExpansion).isEmpty()) {

    CastleMove move =
getRandomMove(node.getUntriedMoves(availableMovesForExpansion));

    int player = currentGameState.getCurrentPlayer();
    doMove(move, currentGameState);
    node.addChild(move, player);
}
```

We then simulate a game played out using random moves until reaching a terminal state.

```
// Simulation
while (!getMoves(currentGameState).isEmpty()) {
    doMove(getRandomMove(getMoves(currentGameState)), currentGameState);
}
```

Lastly, we reach Back-propagation. Here we work our way back up the tree updating each node we traversed with the eventual winner.

```
// Backpropagate
while (node != null) {
    node.update(getWinningPlayer(currentGameState));
    node = node.getParentNode();
}
```

The ‘update’ method increments a ‘wins’ variable within the node if the MCTS player wins – this helps guide the UCB algorithm to eventually select the best move for the MCTS player in the real game state.

3.3 Game State Collection and Storage

Gameplay data had to be collected and stored in order to train the machine learning models. Data is collected from the point of view of one of the players, we will call this player the observed player. The type of player to be observed can be chosen in the set-up of the application. However, all the data collected for machine learning use was from games between the ISMCTS player (observed) and the lowest-card strategy player.

A snapshot of the game state is taken at the start of the observed player's turn. Each row in Table 1 represents one of these snapshots.

id	HAND	CASTLE_FU	CASTLE_FD_SIZE	OP_HAND_SIZE	OP_CASTLE_FU	OP_CASTLE_FD_SIZE	TOP	DECK_EMPTY	WON
14	5,5,7,7,8,8,9,13	8,13,13	3	3	2,2,10	3	3	0	1
15	7,7,8,8,9,13	8,13,13	3	3	2,2,10	3	6	0	1
16	7,7,8,8,13	8,13,13	3	3	2,2,10	3	9	0	1
17	7,7,8,8	8,13,13	3	3	2,2,10	3	14	0	1
18	3,5,5,6,7,7,8,8,9,9,13,14	8,13,13	3	3	2,2,10	3	4	0	1
19	3,6,7,7,8,8,9,9,13,14	8,13,13	3	3	2,2,10	3	9	0	1
20	3,6,7,7,8,8,13,14	8,13,13	3	3	2,2,10	3	11	0	1
21	3,6,7,7,8,8,14	8,13,13	3	3	2,2,10	3	2	0	1

TABLE 1: DATABASE EXAMPLE ENTRIES

Cards are represented as numbers 2-14 (Jack = 11, …, Ace = 14). DECK_EMPTY and WON are Boolean values. The author chose to collect an almost full set of data from the game state with minimal formatting or manipulation. The reason for this was to leave all the data processing to the machine learning scripts, allowing more freedom to try out different methods of processing once the raw data has been pulled from the database. In addition, it also saves any unnecessary computation in the event of the data being processed twice. Storing all the data available, without omitting any on the assumption it is not relevant, is the safest option because until experimenting with the machine learning techniques it is hard to know what parts of the data will be of most use.

As mentioned in the Setup section (see Setup), there are configuration options to export game states for storage. The game engine application supports exporting to CSV files or instead uploading to a MySQL[5] database located on the University network. Should the application fail to upload to the database, the results will be exported to CSV files instead. Additionally, a script was written in Java, separate from the main application, responsible for reading the CSV files and uploading the data to the database.

The automation of this process is extremely useful in order to collect a mass of game states to use for machine learning (see Machine Learning). Several instances of their application were run on the University OpenStack system from a Linux shell. By using the terminal multiplexer Tmux[6], the author could leave the instances populating the database indefinitely.

[5] https://www.mysql.com/
[6] https://en.wikipedia.org/wiki/Tmux

4 Machine Learning

In this section the author describes the steps taken to try to improve the AI employed beyond the ISMCTS implementation. This will include high level explanations of the methods used and how these methods were used in the context of the problem. Results and evaluation from these methods will be left for the next section (see Results and Evaluation).

4.1 Support Vector Machine

The Python library Scikit-Learn[7] (Sklearn) was used in order to train a model to predict if a player would win or lose a game, given a certain game state. A Support Vector Machine (SVM) was chosen due to its ability to generalise from labelled data. Jupyter Notebook[8] was the tool used to write and execute the Python code.

4.1.1 Data preprocessing

Due to the format of the data in the database not being valid input for Sklearn's models, data preprocessing was needed. The data was split into features and corresponding labels. The features are made from all the information gathered about the game state. The labels indicate whether that game state was from a game in which the player won or lost. All the features need to be normalized, so the numeric values fall between 0 and 1.

id	HAND	CASTLE_FU	CASTLE_FD_SIZE	OP_HAND_SIZE	OP_CASTLE_FU	OP_CASTLE_FD_SIZE	TOP	DECK_EMPTY	WON
133	3,5,5,6,11,11	4,10,13	3	0	11,11	3	14	1	1

TABLE 2: DATABASE EXAMPLE ENTRY

Normalisation is achieved by encoding the data. The encoding process is described in Table 3 by taking the entry in Table 2 as input. The encoding function outputs a 1x23 array which is then ready to be used as input to the SVM.

[7] https://scikit-learn.org/stable/index.html
[8] https://jupyter.org/

		Input		Output
	Array Index	Card Value	No of cards in players' hand	Normalised array value
Players Hand	0	2	0	0
	1	3	1	0.25
	2	4	0	0
	3	5	2	0.5
	4	6	1	0.25
	5	7	0	0
	6	8	0	0
	7	9	0	0
	8	10	0	0
	9	11	2	0.5
	10	12	0	0
	11	13	0	0
	12	14	0	0
Players FUCC	-	FUCC index	Card Value of FUCC	Normalised array value
	13	0	4	3/13
	14	1	10	9/13
	15	2	13	12/13
Players FDCC	-	No of cards in player FDC		Normalised array value
	16	3		3/4
Op Hand	-	No of cards in opponents' hand		Normalised array value
	17	0 (4)		0 (4/50)
Op FUCC	-	FUCC index	Card Value of FUCC	Normalised array value
	18	0	11	10/13
	19	1	11	10/13
	20	2	0	0
Op FDCC	-	No of cards in opponent FDC		Normalised array value
	21	3		3/4
Deck Empty	-	Is deck empty?		Normalised array value
	22	YES		1

TABLE 3: SVM DATA ENCODING

4.1.2 Support Vector Machine

SVMs are supervised learning models that analyse labelled data for classification or regression. The author used classification for this problem because the outcomes were binary categories, win or lose. SVMs work by mapping the data to high-dimensional feature spaces. The data clusters and boundaries form between different categories. These boundaries can then be used to classify a given set of new features as either a win or a loss.

The two most important parameters to tune on an SVM is Gamma and C:

"Intuitively, the `gamma` parameter defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'.
The `gamma` parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors." – scikit-learn.org[9]

"The `C` parameter trades off correct classification of training examples against maximization of the decision function's margin. For larger values of `C`, a smaller margin will be accepted if the decision function is better at classifying all training points correctly. A lower `C` will encourage a larger margin, therefore a simpler decision function, at the cost of training accuracy. In other words, 'C' behaves as a regularization parameter in the SVM." – scikit-learn.org

Gamma was tuned by using Sklearn's 'Plot Validation Curves' code[10]. This varies gamma between a set of values; trains and cross-validates an SVM at each value and then plots the accuracy results on a graph for comparison.

The author tuned C using the GridSearchCV method that comes with the Sklearn library. It simply takes a range of parameters and then trains SVMs with all combinations. These SVMs are all cross-validated and the accuracy stored. The combination with the highest accuracy can then be obtained. However, overfitting can occur and skew results. Overfitting occurs when the model does not generalise to data points outside of the training data. Instead it 'memorizes' training data rather than learning the data set's trends.

These two methods had to be repeated and fine-tuned to get closer to the optimum parameters.

4.2 Convolutional Neural Network

Keras, also a machine learning library for Python, was used to set up and train a neural network. The author used Google Colab as the tool for writing and executing the code. The primary reason for moving from the University OpenStack to Google is because Google provide much more GPU power. A consequence of this meant having to copy the database from the University MySQL server to Google's CloudSQL due to the limitations of the University database being within the secure University network.

[9] https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html
[10] https://scikit-learn.org/stable/auto_examples/model_selection/plot_validation_curve.html

4.2.1 Data preprocessing

As with Sklearn, neural networks within Keras also need the data to be preprocessed. However, neural networks provide better results when fed image data. This meant that game state entries stored in the database needed to be converted to images first. To reduce the complexity of the input, the author reduced the information to just the player's hand, face up castle cards (FUCC) and the number of face down castle cards (FDCC) plus the opponent's FUCC and number of FDCC. This decision was made with the aim of preventing the model from overfitting due to having too many inputs. The data was converted to an image by using a binary 2-dimensional array.

Using the example entry shown previously in Table 2, it is represented by the 4x41 image in Figure 8.

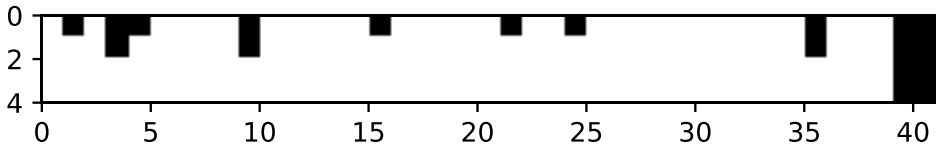

FIGURE 8: GAME STATE IMAGE REPRESENTATION

The data is split into a training set and a test set, using 80% and 20% respectively. The training set was used to train the model, while the test set is used to test the accuracy of the model. Due to the large amount of data collected, a 20% test set was over 200000 entries and, therefore, enough to confidently test the model.

4.2.2 Creating a convolutional neural network

The convolutional neural network (CNN) created has five convolutional layers each followed with max-pooling layers before a final fully connected layer. Krizhevsky et al (2012) found a similar structure to be effective on the ImageNet set [8]. Dropout layers were introduced after the model showed signs of overfitting to the training data. Like the SVM approach, the CNN also categorises game states to a win/loss. This categorisation is done in the final fully connected layer via a sigmoid activation function. Once trained, the CNN was exported to use within the game engine. The summary of the CNN output from Keras is displayed in Figure 9.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 4, 41, 5)          50
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 2, 21, 5)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 2, 21, 10)         460
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 11, 10)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1, 11, 10)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 11, 20)         820
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 1, 6, 20)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 1, 6, 20)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 1, 6, 40)          3240
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 1, 3, 40)          0
_________________________________________________________________
dropout_3 (Dropout)          (None, 1, 3, 40)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 3, 60)          9660
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 1, 2, 60)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 120)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 242
=================================================================
Total params: 14,472
Trainable params: 14,472
Non-trainable params: 0
_________________________________________________________________
```

FIGURE 9: CNN STRUCTURE

4.3 Custom Evaluation Function

In order to import and make use of the created CNN, Java libraries ND4J [11]and DL4J[12] were used. ND4J allows the creation and manipulation of n-dimensional arrays fundamental to neural networks. DL4J provides the classes and methods needed to import and query models. When the ISMCTS player is initialised, so is an *Evaluator* class. This is then called via the *evaluate* method and returns a Boolean value - true if the player is expected to win, or false if not.

[11] https://nd4j.org/
[12] https://deeplearning4j.org/

This evaluation can be used in a multitude of ways. Currently, only one type of ISMCTS player with a custom evaluation function has been added. This works by stopping the simulation of games at a given depth, and then using the state the simulation ends on to query the model. The result from the query is then backpropagated up the tree, as in the standard ISMCTS implementation. The aim of this was to reduce the time it takes for the ISMCTS algorithm to run, because it will not have to randomise gameplay for as long as it takes to reach a terminal state. It was also thought that it could improve the effectiveness of the algorithm, because the values that are backpropagated could be more accurate in rating considered moves, if they originate from higher up the tree. Higher up the tree means there have been fewer random decisions moving away from the originally considered state.

For other possible uses and implementations of custom evaluation functions, see Future Work.

5 Results and Evaluation

5.1 AI Player Effectiveness

This following section will evaluate the effectiveness of the two implementations of the ISMCTS algorithm as well as the benchmark player with a lowest-card strategy. This has been done by playing many simulations of these players against each other and recording the results, shown in Table 4.

A \ B	Lowest-card strategy	Standard ISMCTS	Custom Evaluation ISMCTS	Random Decisions	Human
Lowest-card strategy		22.5% 21540 games	30.4% 20 games	91.2% 1000 games	
Standard ISMCTS	67.5% 21540 games		46.7% 30 games	98.0% 100 games	60.0% 5 games
Custom Evaluation	69.6% 23 games	53.3% 30 games			
Random Decisions	9.8% 1000 games	2.0% 100 games			
Human		40.0% 5 games			

TABLE 4: PLAYER WIN RATES (A BEATS B)

5.1.1 Lowest-card Strategy

The lowest-card strategy beat a player making random decisions in 91.2% of the 1000 games played – using the binomial distribution the author calculated the probability of this happening by chance to be $p<0.0001$. Therefore, this result is highly statistically significant in rejecting the null hypothesis that the win rate for the lowest-card strategy would be no higher than random decisions. This suggests that the lowest-card strategy is at least somewhat effective in the game of Castle.

5.1.2 Standard-ISMCTS

The results between Standard-ISMCTS (S-ISMCTS) and the random player shows S-ISMCTS winning 98% of games; an improvement on the lowest-card strategy results.

The null hypothesis that S-ISMCTS would be no more effective than the lowest-card strategy is rejected by the results between the two strategies, where S-ISMCTS is winning just over two thirds of games – this is also highly statistically significant with the probability of this happening by chance being $p<0.0001$.

In a separate analysis, an S-ISMCTS player played against 3 lowest-card strategy players and won 29.4% of games over a sample of 1000 games. This provides evidence it is still a more effective strategy even when in a larger game. The probability of this happening by chance is $p<0.019$, also statistically significant.

5.1.3 Evaluating-ISMCTS

5.1.3.1 The Model

After many different iterations of model structure, the best accuracy score produced was ~73%, shown in Figure 10. Typically, this is not regarded as a high score, although in the context of this project it was sufficient to produce a successful player. Two main reasons for struggling to reach a higher accuracy score could be: (i) because of the randomness in Castle and (ii) the fact that the model is trying to predict a winner. The possibility of predicting a winner accurately is low as the winner has not been determined at a given point, with chance still playing a large part in the outcome of the game.

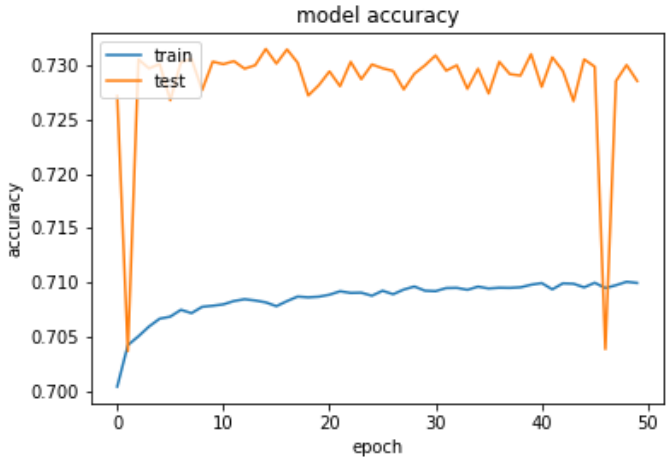

FIGURE 10: CNN ACCURACY

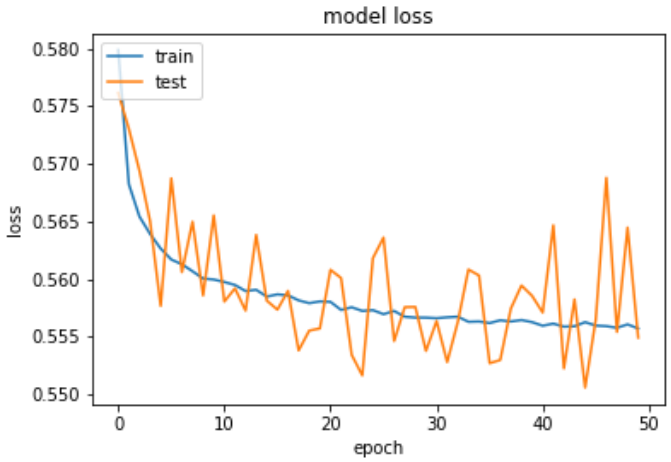

FIGURE 11: CNN LOSS

5.1.3.2 The Player

One of the first observations after running simulations with the Evaluating-ISMCTS player (E-ISMCTS) was that games took at least ten times longer than with S-ISMCTS. This is likely due to the ND4J and DL4J libraries' high use of memory and then Java's garbage collector slowing down computation as it clears unused memory. This is only an assumption and due to time constraints, there was not enough time to do an investigation to be sure of the source of the problem and to produce a solution. Therefore, it was only possible to run a limited number of simulations with this AI.

The null hypothesis that E-ISMCTS would be no more effective than the lowest-card strategy is rejected by the results between the two strategies, where E-ISMCTS, similarly to S-

ISMCTS, is winning just over two thirds of games – this is also statistically significant because the probability of this happening by chance is $p<0.017$. Ideally, more simulations should be run in order to gain a higher significance level.

In summary, the E-ISMCTS player has performed very similarly to the S-ISMCTS player over the limited trials available. The author suggests that with more time to tweak the implementation, the win-rate could be further improved and push ahead of the S-ISMCTS.

5.1.4 Human interaction

The author played 5 games against the S-ISMCTS player, managing to win in 2 of them. Due to the very small sample of games and the use of only one human, no valid conclusions can be made on the effectiveness of the S-ISMCTS player against human players. However, it provides the opportunity to evaluate the S-ISMCTS player from a gameplay perspective. The author observed that the AI made intelligible moves that did not look uncharacteristic of a human player's choices. It was apparent that the S-ISMCTS plays a very similar strategy to the lowest-card strategy throughout the game. Although, it has already been established that it is playing a better strategy (winning 67.5% of games) than the lowest-card strategy but specific differences were not noticeable from the limited games played. The only observed difference was in the choice of FUCC – a decision point that could have been further analysed given more time. The end goal mentioned at the start of the report was to have produced an AI capable of competing against a human player. By winning 3 out of the 5 games, it is reasonable to surmise that the S-ISMCTS player can compete with a human player. However, the full extent has not been explored nor demonstrated beyond doubt. In order to gain further confidence in the strength of the S-ISMCTS player, hundreds more games would have to be played between the AI and various human players (see Future Work).

Although it was not the aim of the present project, it is worth noting that the human experience of playing the game against the AI did not feel authentic. The author suspects the reason for this could be down to the user interface. Time limitations meant the only UI was through the terminal, restricting the way the user views the game state, and their ability to choose actions in an interesting way. The way in which players pick moves, i.e. via a numbered list of options, is uninspiring. Without any graphics or animation, the game feels uninteresting and very quickly monotonous.

5.2 Game engine

If this project were to be undertaken by the author again, unit tests would be produced during development to be more certain that everything works as intended. In this case however, the game engine's soundness has been primarily evaluated by observing games between players to check that it plays correctly: Once the game engine was finished this was always observed to be true. A method was also produced that periodically checks the validity of the cards present within the game state, to ensure there is a full deck. This proves that no cards were disappearing from the game or that extra cards were being added. Between close observation and the validity checking method, it can be confidently assumed that the game engine performs as intended.

6 Future Work

With the main parts of the project explained and the results evaluated, the author now moves on to cover future work. This mainly focuses on exploring further improvements to the E-ISMCTS, as this was the main focus towards the end of the project. However, we will also briefly examine how the core ISMCTS algorithm could be modified to include game specific knowledge as well as discussing the possibility of this implementation of Castle being released as marketable game.

6.1 Improvement of the Evaluation Function

One of the main issues found with the E-ISMCTS algorithm is that it was extremely slow to run simulations with – sometimes over ten times slower than S-ISMCTS. This could be improved in the future to allow more efficient evaluation as the algorithm is further improved. Given more time, investigation of memory usage by the DL4J and ND4J frameworks could identify any issues caused by Java's garbage collector. After a small but conclusive amount of testing, the author found that querying the model within Java took longer than in Python. Therefore, the evaluation function could be stripped out of the main application entirely and produced in Python as a stand-alone system. This Python program could still be run locally for very low latency.

There are also different approaches to the inner workings of the evaluation function that could be explored. In the current implementation, all the model's predictions are given equal weighting. However, this may not be the best approach because the predictions vary in certainty. Instead, the activation value from the model's final layer could be backpropagated up the tree as this value can be used to assess how confident the model is with a prediction. For example, if the activation value were 0.6, it is proposed that 0.6 be backpropagated instead of 1 (a win) as in the current implantation. Another adaptation would be to only accept predictions when the model is at a certain level of confidence, e.g. < 0.4 or > 0.6. In the instance of the model not being confident in its prediction, the algorithm would then carry on simulating for a given number of turns before querying the model again. In principal, this means that the algorithm should run as S-ISMCTS unless the model is confident enough in its predictions.

Furthermore, this project has focused on exploring Support Vector Machines and Convolutional Neural Networks but there may be more suitable ML techniques available. Sklearn provides many more functions that may be worth exploring such as Random Forests as well as there being additional neural networks architectures to experiment with.

6.2 Improvement of ISMCTS

Game specific knowledge could be added as further improvement to the ISMCTS algorithm. This could give the algorithm the ability to 'remember' cards that have previously been played by themselves and opponents in order to make better informed decisions by narrowing the information set down.

An issue for ISMCTS within the game of Castle is the game's large state space. This could mean the simulation stage is not sampling a very large proportion of the space. One way to combat this would be to add bias to the simulation stage. This would direct simulation to more likely states within the space in order to more accurately model game play throughs. SD James [10] demonstrates how this approach can provide mixed results.

6.3 Development of a Marketable Game

As shown in the Evaluation and Results section, the current AI is already at a level that is 'likely' to challenge human players. However, if more human tests were carried out to confirm this, it would open the possibility of a different approach, i.e. developing the parts necessary to release the game to the public to play. These tests would need to cover participants of varying abilities. A releasable game would require development of a GUI and network capabilities, both of which are viable, albeit time consuming to produce.

7 Conclusion

The aim of this piece of work was to design, build and improve a Castle AI player as far as possible within the given time frame. In order to achieve this, it was important to spend time designing and implementing a game engine with a modular architecture. This time investment was repaid later in the project by allowing easy configuration changes when altering and testing the AI players. While a GUI was outside of the scope of this project, a clear and easy to use console-based text interface was produced. It was useful for displaying real time play between AI players and for the human player to use when playing against the AI. However, being constrained to the terminal meant it did lack aesthetic qualities and was uninspiring for the human to interact with.

The game engine code is written clearly and concisely, allowing the intended audience to understand the way it works when read in combination with the UML diagrams and brief explanations given in this report. Given more time the author would have preferred to make further refinements to the code base. However, these refinements are not essential for the overall aim of this project.

Moving on to the machine learning part of the project, the game engine ran and exported results effectively, collecting over a million game states as training data. Having never used Linux before this project, making use of the University Linux labs and Openstack Linux Machines has prepared me for using the Linux operating system in industry after University. Experimenting with some of Google's cloud computing modules such as Google Colab and Cloud SQL has also been a good learning experience. Using these technologies to write and run Python scripts remotely provided the computation needed to effectively train and test two types of machine learning models, Support Vector Machines and Convolutional Neural Networks.

It was disappointing to achieve a model accuracy of only 71%. However, this model score did not translate into a poorly performing player. On the contrary, the E-ISMCTS player beat the lowest-card strategy player in ~70% of games – very slightly higher than the S-ISMCTS player. Both ISMCTS players performed well against the benchmark lowest-card strategy player and the S-ISMCTS won 3 out of 5 games against a human player. These results are evidence that the AI players have been improved to a suitable level that fits with the aim of this project.

Although this project was successful against the chosen benchmark player, limited human trials prevented proper comparisons with human play. Neither of the AI players built are 'perfect players' and the author believes that with more time the E-ISMCTS player could have been vastly improved, both in terms of speed and performance, in order to give human players a more life-like game experience and challenge.

8 Reflection

A definite point for reflection is the time management of the machine learning exploration. My model performances peaked at ~73% accuracy (see 5.1.3.1) early in the process and because I assumed this was not high enough, I spent weeks trying to improve it with little success. However, once I loaded the model onto the E-ISMCTS player, it showed promising performance, suggesting that the time would have been better spent improving the player as opposed to the model. Reflecting upon this, I should have focused more on testing and refining the player instead of the model. I believe that my underlying assumption that basic time management and planning is not necessary when working on a solo project caused me to make this mistake.

If I were to do this project again, I would factor in time to write full unit test coverage. This would have helped to prevent the situation I found myself in; collecting 4 million data points over the space of weeks before I realised that the ISMCTS player had a bug which caused its win rate to drop by ~15%, causing the data to be inaccurate and tainted. Had I found the bug earlier I would have been able to collect many more data points of higher quality to train the models with. Originally, I had thought that unit tests are only necessary for larger applications with multiple developers. However, carrying out this project has taught me that you cannot be sure everything is working properly even if you know the application well and it is all your own code.

9 References

[1] Campbell, M., Hoane Jr, A.J. and Hsu, F.H., 2002. Deep blue. *Artificial intelligence*, *134*(1-2), pp.57-83.

[2] Togelius, J., 2015, November. AI researchers, Video Games are your friends!. In International Joint Conference on Computational Intelligence (pp. 3-18). Springer, Cham.

[3] Kocsis, L. and Szepesvári, C., 2006, September. Bandit based monte-carlo planning. In European conference on machine learning (pp. 282-293). Springer, Berlin, Heidelberg.

[4] Jakubik, J., 2017, September. Evaluation of hearthstone game states with neural networks and sparse autoencoding. In 2017 Federated Conference on Computer Science and Information Systems (FedCSIS) (pp. 135-138). IEEE.

[5] Cowling, P.I., Powley, E.J. and Whitehouse, D., 2012. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(2), pp.120-143.

[6] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S., 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), pp.1-43.

[7] Cowling, P.I., Ward, C.D. and Powley, E.J., 2012. Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(4), pp.241-257.

[8] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

[9] Nyholm, S. and Smids, J., 2016. The ethics of accident-algorithms for self-driving cars: An applied trolley problem?. *Ethical theory and moral practice*, *19*(5), pp.1275-1289.

[10] James, S.D., 2016. The effect of simulation bias on action selection in Monte Carlo Tree Search (Doctoral dissertation).