UNIVERSITY OF THE WITWATERSRAND

WITS UNIVERSITY

School of Computer Science and Applied Mathematics
Faculty of Science
HONOURS RESEARCH REPORT

An Investigation of AI Tree Search Methods and Their Effectiveness at Playing the Card Game Gwent

Jason Chalom 711985

Supervisor(s): Professor Clint Van Alten

November 2017, Johannesburg

An Investigation of AI Tree Search Methods and Their Effectiveness at Playing the Card Game Gwent

Jason Chalom 711985

***Abstract*—The aim of this research is to compare AI tree search methods when applied to the game Gwent. Gwent is an interesting domain because it is non-deterministic and has sub-domains which require their own search operations to be performed when decisions are made. Is Gwent random in nature or does the use of strategy affect the outcome of a game? Using empirical analysis on three random agents (and four more intelligent agents) it is shown that whilst Gwent has some random attributes the game itself is not random. Agents which are more geared towards strategic choices (such as MCTS) outperformed all other agents. It is also shown that the MCTS agent is the best agent for playing the game of Gwent but only when its parameter which controls the number of iterations is sufficiently large.**

I. Introduction

The textbook, Artificial Intelligence: A Modern Approach [15] describes games as having been an area of interest for humans since the beginning of organised societies. A game is where two players or agents are in competition in some domain with a defined set of rules. Research in artificial intelligence (AI) has always gravitated towards games due to the complexity and abstractedness they present and yet they have clearly defined rules which make them easier to program and study than other abstract real-world problems such as physical sport. The applications of AI in games can lead to developments in other more serious fields such as applications in economics.

Not all games are the same. Artificial Intelligence: A Modern Approach [15] describes many games such as deterministic games where the outcomes are known, turn based games where each player must wait their turn, two-player games and zero-sum games where in a multi-player arena only one player can win as whatever gains (wins) one player gets the other player must then lose out on. A game with perfect information means that both players know full-well what the outcomes of all previous moves were.

Games are an interesting area of study because of how difficult they can be. Many games are computationally difficult to solve - both in complexity and time. Some games can have so many states that would have to be computed that it is in-feasible to calculate all of them within the time constraints of a game where inefficiency could have penalties. Also due to the very nature of deriving game decisions and paths for future moves, having branching factors which depend on the types of games being solved will produce exceedingly large solution spaces. This means that strategies which employ smart use of resource constraints in a given domain are required to solve these problems. [15].

A. Problem Domain: The Card Game Gwent

The Gwent game guide [7] describes Gwent as a two-player turn-based card game based on a medieval (fantasy) battle field. It is a game played to the best of three rounds where the players use the same hand of cards drawn from their respective shuffled decks prior to start of play. Each player creates a deck from the total cards they have based on a certain faction and they choose cards whilst following a set of rules which govern how theses decks should be constructed. There are a few different rule sets and as new cards are introduced to the game new rules may be added. This game is interesting because it is a zero-sum game with partial information. The players both know the moves which came before but they do not know the opposing players hand. Gwent has a number of interesting rules such as there being a card which allows a player to select a card from their discard pile and playing it again. Another interesting card is one which the player plays on the opposing side of the board. Here the card acts against the player who played the card in return for two new cards from the players deck.

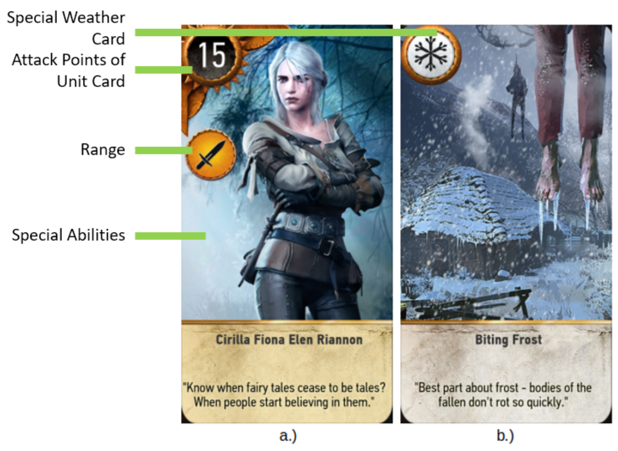
Special Weather Card
Attack Points of Unit Card
Range
Special Abilities
15
Cirilla Fiona Elen Riannon
"Know when fairy tales cease to be tales? When people start believing in them."
Biting Frost
"Best part about frost - bodies of the fallen don't rot so quickly."
a.)
b.)

Fig. 1. Example of a unit Gwent Card (a) and a weather/special card (b). [7] [1]

B. Aim of Research

Gwent is an unknown domain in terms of how different variants of tree search algorithms such as MCTS may perform. A search through existing literature has not found any research which applies these methods to this exact problem. Gwent is also a relatively new game which is still being actively developed [1] which may explain the lack of its use in existing literature. I wanted to prove that the game of Gwent is not a purely random game of chance but rather that a well implemented strategy will determine the outcomes of a game. If this is true then random agents should perform poorly when compared to more advanced AI agents.

1) Hypothesis: My hypothesis states that:
The game of Gwent is not random in nature and therefore AI agents will have higher win rates when compared to random agents.

II. Background and Related Research

A. The Rules of Gwent

Gwent is a 2-player turn-based game, with three rounds [7]. The game is the best of three rounds. Each player must have a chosen deck which consists one leader card, twenty-two unit cards minimum, and ten special cards maximum. Unit cards which are hero cards, cannot be revived or effected by any other card. Ten random cards from the chosen deck are drawn at the start of the three rounds for both players. Each player takes a turn placing one card on the board. The board has three rows on both sides. Certain units can only be placed on certain rows. A player can also at any turn decide to pass the remainder of the round. The winner of a round is decided when either both players run out of cards, both players pass the round or a combination of the two. A round win is calculated by adding up the strength of all unit cards whilst taking into account the effects of the special cards.

Special Cards:

- There are four types of weather cards, one clear and one for each row of the battlefield (on both sides). The weather cards reduce all unit cards except hero cards to attack points equal to one (unless commanders horn is in effect then attack points of two).
- The commander's horn doubles attack points of all normal unit cards in that row.
- Scorch destroys the strongest valued cards on the board (i.e. the highest number, any card with that number is discarded)
- Decoy allows a player to take a card back into their hand.

Abilities of Some Unit Cards:

- Spy units go to the other players side in return for drawing two new cards from the player's deck into their hand.
- Medic allows you to look through the player's discard pile and revive one unit card (must not be a special or hero card).
- Tight bond doubles the attack points of a group of cards of the same type of unit (with this ability) when they are placed next to one another (The doubling is calculated on the combined strength of every card in the group).

B. Literature Review

1) AI Tree Searches: The basic idea of using tree searches to make decisions in game problems is to model the problem (or domain) as a set of decisions which are in the form of a tree. Each node of the tree as described by Artificial Intelligence: A Modern Approach [15] are specific game states and each edge of the tree is a certain decision or a certain path which can be taken from the parent node to get to the children nodes or the next set of possible game states. The root of the tree is either the current state of the game or the initial state depending on the implementation of tree search. A terminal node on a tree is a node where either the game has reached a stop (or end) state such that no further children can be created or a limiting condition has been met which limits the depth of the tree.

2) Tree Pruning: One notable problem with the approach of generating trees based on possible domain states of a game is the sheer size and complexity which can emerge from many games such as Gwent and its contemporaries. Edwards et al. [10] and in chapter 5 of Artificial Intelligence: A Modern Approach [15] both describe how a tree constructed from all possible game states is wasteful as there are many branches which need not be considered. The growth of these trees are exponential in nature (based on the size of the domain being modelled and searched) which creates a problem of time and complexity when dealing with the implementation of such an AI. In such cases these paths can be removed from the tree because they have no effect on the outcome of the tree search. This can greatly reduce the size of the tree before a complete search is done.

Another approach as described by Chaslot et al. [9] is to only invalidate a node (and therefore the rest of the path that follows) which is determined to be unreliable (or as a node not visited often in a tree search) and then these nodes can be re-evaluated and selected later (based on the search algorithm utilised).

3) Minimax Algorithm: Having created a tree of possible paths and game states an algorithm is needed to help determine which is the most optimal path for the AI to take in order to maximize its chances of winning the game (thereby minimize the chances of the other player from winning). This sets up the AI player as the MAX player and the opponent as the MIN player. One of these algorithms is called The Minimax Algorithm. Artificial Intelligence: A Modern Approach [15] describes this algorithm as a method of finding an optimal set of strategies. This is done by placing a value on each node of the tree which is called the Minimax value. This value is a numerical representation of how useful that decision is for the MAX Player i.e. the MAX player favours a higher value and the MIN player favours a lower value. In this set-up, for each decision that MAX makes, MIN must also make a decision and these two decisions together make-up one move on the tree - this is the basis of a multi-player game. These "half-moves" are defined by Artificial Intelligence: A Modern

Approach [15] as a ply. The Minimax decision is the decision which leads to the optimal outcome for MAX from the root node. This algorithm will recursively generate the Minimax values for every child node in the tree by using an objective function which determines the numerical outcome of that specific node. Then the algorithm back-propagates up the tree updating the Minimax value of the parent nodes until root is reached where it terminates.

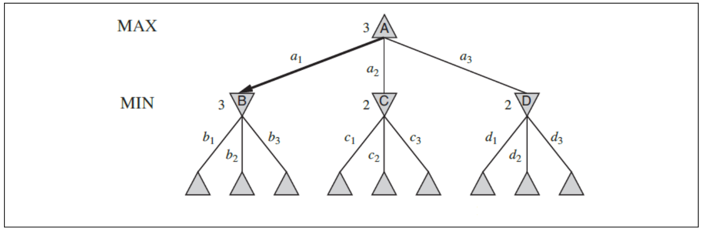

Fig. 2. Example of a Max tree where the Max nodes are the upright triangles and the Min nodes are down facing triangles, taken from Chapter 5 of Artificial Intelligence: A Modern Approach [15].

One note which is made by Artificial Intelligence: A Modern Approach [15] on this algorithm is that there is an inherent assumption that both MAX and MIN play optimally. Also the Minimax algorithm in its most basic form will perform a full depth-first search of the tree which means that it will have an exponential growth in time complexity which corresponds to the exponential growth of the tree based on the domain space. The space complexity of this algorithm and the time complexity are determined by the depth of the tree and number of moves at each layer of the tree.

4) Alpha-Beta Pruning: The Alpha-Beta Pruning Heuristic was conceived as an extension of the Minimax algorithm. Edwards et al. [10] states that it uses partial information from the tree to decide which branches on the tree are no longer needed for Minimax. These removed branches do not affect the final decision determined by Minimax. The method uses the fact that if a move is found to be worse than a move evaluated previously (either for the MIN player or the MAX player), than that move does not affect the outcome of the search. Edwards et al. [10] highlights the importance of the ordering of the possible set of plies from the current node (position) because this heuristic achieves the best results when the optimal decision is located at the beginning of the set of positions. To this effect the Alpha-Beta Heuristic can at most cut a search trees growth rate in half, which implies that this heuristic allows a computing system to search twice the depth within the same time complexity of the basic Minimax algorithm. Edwards et al. [10] proves this by induction in their technical memo. This performance increase is a powerful first step in using Minimax in a real world domain.

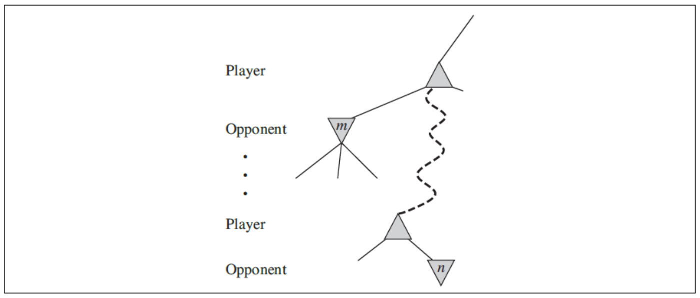

Fig. 3. General case of alpha-beta pruning where if m is a better move than n, n will be removed, taken from Chapter 5 of Artificial Intelligence: A Modern Approach [15].

5) Monte-Carlo Tree Search Algorithm: The basic idea of MCTS as described by James et al. [16] and Browne et al. [6] is that it combines the idea of a tree search with the idea of simulating parts of the domain space. Monte Carlo simulations are used to make approximations of some problem based on random sampling of the problem domain [6]. Browne et al. [6] describes MCTS as a family of heuristic optimisation methods for making optimal decisions in certain domains and game spaces. The basic operation of these methods sample the domain space (in similar fashion to MC methods) by simulation which is then used to build a search tree based on this sample of the domain. These methods work well with difficult problems such as hidden domains and problems where other techniques have been unsuccessful since they are stochastic in nature.

Browne et al. [6], James et al. [16] and Chaslot et al. [9] all describe a basic construction of a MCTS as having four major steps: Selection, Expansion, Simulation (also known as roll-outs [16]) and Back-propagation The selection step is when the tree is traversed from the root node through to the most urgent terminating nodes of the tree - which is decided by some predefined policy. Chaslot et al. [9] describes four ways of determining the best child node as the final move selection. These look at how often a child is visited, the value of the respective child and some combination of these and other parameters. The expansion step is when one or more children are added to the selected terminating node. A simulation is then run on the new expanded nodes to determine what the outcome of those actions could be in the future. The simulations are also governed by a policy, which in the most basic case is to perform a random move. Finally this result is back-propagated up the selected path to update the outcomes of this path (up to the root node). This process is repeated until a termination condition is met such as a limit on the number of iterations, or the size complexity of the search tree.

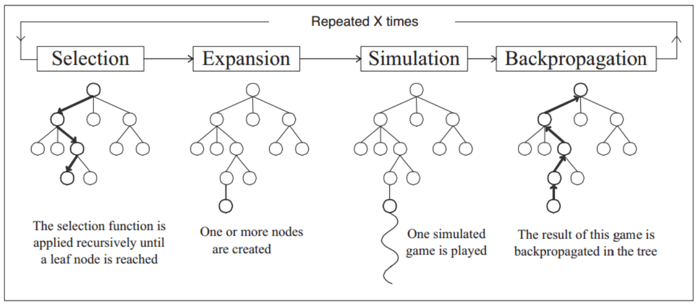
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Fig. 4. Steps of the Monte-Carlo Tree Search, taken from Chaslot et al. [8].

6) The Use of Expanded Techniques with MCTS: Browne et al. [6] describes many different optimizations and enhancements to the basic MCTS and shows that there are many variants of MCTS which change different aspects of the four components of the algorithm.

Upper Confidence Bounds for Trees

Bandit based methods as used with MCTS are a modification of the selection phase of MCTS [6]. Browne et al. [6] describes bandit-based problems as a set of problems where there is an exploration-exploitation dilemma. This dilemma is that there is contention between the explorations of deeper variant moves with high win rate versus moves with few simulations associated with them (for the case of MCTS). Kocsis et al. [12] applied a specific bandit algorithm for MCTS, known as UCB1. This application of a bandit-based policy for MCTS is known as the UCT algorithm. Here the algorithm uses random probabilities based on a policy mapping of what the next move that will be decided (as is generated in the expansion step). These generated moves are then checked against a regret (objective) function with the aim of minimizing the regret. The regret function used calculates the loss based on how many times the policies did not use the best possible path. The UCT algorithm, treats every child node of the move selection which has been explored as separate multi-armed bandit problems. The arms of each bandit problem are the available moves on the sub-tree and the child nodes are the weighted rewards of the paths calculated using the bandit-algorithm. During testing Kocsis et al. [12] found that UCT appeared to produce a lower error than AB, MCTS and Monte-Carlo planning with Minimax value update (A hybrid approach to tree search).

James et al. [16] have however discovered that UCT operates far better when there is a preservation of the optimal decision rankings. This is very similar to what Edwards et al. [10] found with AB pruning. Caution must then be taken when using the UCT method as James et al. [16] found that if the distributions of the random simulations have small variances than UCT can become sub-optimal and even worse than random. James et al. [16] noted that there has been a vast amount of work in researching alterations (such as UCT) in the selection and expansions steps of MCTS but less work has been done in the simulation step.

Progressive Strategies for MCTS

Chaslot et al. [9] suggested two progressive strategies which leave out infrequently visited nodes that can be selected and examined later. This is done using available information and some heuristics (which are computationally expensive) to select nodes which are visited often and invalidate the rest.

Progressive bias as presented by Chaslot et al. [9] and described in depth by Browne et al. [6] is a method which attempts to replace unreliable nodes (not visited often) with some heuristic function which takes in this suspect node as its input. This method works on the selection and expansion steps of MCTS. As the node is visited more (as the number of iterations of MCTS increases), the method will ensure that the heuristic's influence decreases. The heuristic may also only be applied after some fixed number of visits have passed for each node under question. This means that initially the MCTS functions as normal, after sometime the selection algorithm is applied to each node so the children with the highest heuristic values get chosen first due to there not being enough simulations to make an accurate prediction. As the number of simulations grows, the impact of both the heuristic function and the simulations are balanced. As the number of simulations keeps growing a boundary will be reached (determined by the heuristic function used) where the influence of the heuristic function will decrease until it has little effect whereas the effects of the simulated results now have a high impact on the outcome and the behaviour becomes similar to the basic approach of MCTS.

Progressive unpruning is another method detailed by Chaslot et al. [9] and described by Browne et al. [6] as a soft pruning method where possible decisions are ignored but may be evaluated later by using heuristic knowledge to reduce the size of the tree. This algorithm works when the number of simulations of some node equals a predefined threshold, a heuristic is then used to invalidate the less likely to be visited children nodes such that only the most likely nodes are selected. As the number of simulations gets bigger and further from the threshold, the children nodes should be progressively unpruned so that they may be evaluated. This reduces the number of sub-branches to deal with during the lower set of iterations of MCTS. Chaslot et al. [9] has shown that this method helps with the time-complexity of the MCTS whilst improving the accuracy for a lower number of samples (simulations) when using MCTS. This provides the benefit that better results with lower error can be produced in a smaller time complexity than the vanilla MCTS.

Using Biasing Techniques with MCTS (Expert Knowledge)

Whitehouse et al. [18] proposed a method for initially biasing MCTS so that their commercial AI would seem more human to human players of their game. Their algorithm ISMCTS guesses potential game states and generates a game tree from the statistics of many potential game states by running simulations. Each iteration of the algorithm generates a new tree. They found that the injection of heuristic knowledge in MCTS was easier to implement than an expert heuristic system. They limited ISMCTS to only make decisions which can be contained in one round of their game because this algorithm is unable to look ahead to future rounds. The approach taken was to bias the initial selection step of MCTS and as the algorithm runs the applied bias tends to zero thereby avoiding the issue of weakening the AIs ability to take the best move for that round. The bias values are derived from existing knowledge. The tendency of the bias to go to zero is enforced during the back-propagation step when the updates of the node values which have a bias greater than 1 get their rewards increased whereas bias values less than 1 get reduced rewards. Whitehouse et al. [18] found that the MCTS chose the wrong move less than five percent from the tests conducted which was expected due to the random nature of the algorithm.

There is however contention between Whitehouse et al. [18] and James et al. [16] as the former expects the injection of expert knowledge (bias) into MCTS will decrease the accuracy of the algorithm in choosing the most optimal decision whereas James et al. [16] found that the choice of bias is important in making sure that the performance of MCTS is not negatively affected and in some cases MCTS can perform better with bias added to the algorithm. There are some game problems where there is so little information supplied that running enough simulations to overcome this gap is in-feasible - a solution here would be to use an informed simulation.

7) The Idea of Hybrid Tree Searches: Baier et al. [4] highlights that Minimax search techniques are very good at selecting the most optimal strategies but at the cost of performance whereas MCTS is very good at dealing with large and complex search domains (such as incomplete information) but at the cost of error - in some cases missing critical decisions which Minimax would not miss. Baier et al. [4] puts forward different areas of MCTS which can be modified to use the Minimax search. The MCTS used in their experiments is the UCT form of the algorithm. A method which Baier et al. [4] described in the literature is to merge MCTS and Minimax is to use shallow (1-ply or 2-ply) searches at every simulation executed in MCTS - this however assumes the existence of an objective function and that the problem does not have incomplete knowledge.

Beam Monte-Carlo Tree Search

Baier et al. [3] describes BMCTS as a hybrid tree search where MCTS is combined with a beam search down the decision paths. A beam search is a search where more promising nodes are expanded whilst other nodes get rejected. This is done to reduce the memory and search complexities of a problem. It this hybrid approach the beam search acts as a pruning device for the MCTS tree. The pruning takes place during the back-propagation phase. The condition of pruning used in this approach is the number of simulations that have run through a node. The algorithm will keep the most visited node, remove the least visited nodes and the nodes at depth d will no longer be created. Nodes beyond depth d will still continue to be explored. Only the retained nodes are taken into account which means that a sub-optimal solution may be found but the size of the potential tree is drastically reduced.

8) Testing Methodologies: James et al. [16] defined domains specifically for their testing purposes. They used grid world and finite MDPs to produce less complex problems in which to conduct their experiments. They then ran the different algorithms and a basic random agent over these domains to get their results which they then took averages of by running thousands of the same experiment sets. Whitehouse et al. [18] had a set domain in which they had to work with. Here testing was done using AI-versus-AI simulations and using a select number of real people to beta test different versions of their AI. Chaslot et al. [9], Browne et al. [6] and Kocsis et al. [12] used many domains (2 or more) to run their experiments and to also see if the domain space itself could cause these different methods to behave differently.

III. Research Methodology

A. Introduction

The proposed methodology attempts to uses similar strategies as seen in the literature under Testing Methodologies (section II-B8). An example is using AI-versus-AI simulations to find which AIs perform better than other AIs.

B. Restrictions Placed on the Domain

A sub domain of Gwent must be chosen in order to limit the scope of this research so that it will be feasible in the allotted time-frame. The chosen decks were limited to two decks which have cards already chosen so that the decks have similar attributes. The AI only decides on choices in the current hand and does not construct any decks prior to the game. These decks keep some of the special rules of the game such as being able to revive discarded cards but will not contain cards which can target specific opponent cards. The leader cards and their abilities will also be ignored. All AI (player) responses are limited by an iteration limit as this ensured that the many experimental games that need to be run could be done in a realistic time-frame.

C. Experimental Method

A game supervisor was built which produces the domain and enforces the rules of the game. The AIs also use it as a framework for their simulations and interactions with the game. Several AIs using versions of MCTS were made. A unit testing framework was also utilised in order to check that the implemented rules work as expected. Random AI agents who play random moves from a valid move list (for that game state) were also produced. All the AIs must be run against the random agents to determine if they perform better than random on the domain. A tournament setting will then be setup where different AIs will be pitted against each other in a round-robin style to find which algorithms perform the best. This will be done in rounds of experiments where AI opponents will be set so that a large enough set of results can be collected and analysed to determine which AIs are better for each group of experimental rounds. Each individual experiment in a round will pit the same two AIs but the game state that is set will be randomly generated based on the available cards. Each experiment is run twice with the players swapped (which means swapping the decks as they are associated with players) - this is done to ensure fairness of the matches. Three pseudo-random agents are used to see what effect the PRNG has on the results.

D. Implementation

The programming for the game supervisor and all the AI methods was done in C++. As described by Bronson et al. [5] C++ allows for the use of inheritance techniques which allows for concepts such as the base tree implementations to be shared between the different AI implementations. C++ is also a compiled language which provides lower level access to a computer systems resources which allowed me to make code which runs faster than equivalent code in an interpreted language's virtual environment. Another useful implication of using C++ is that many advanced parallel computing libraries such as OpenMP are extensions of C and C++ [17]. Many parts of the testing system has been parallelised in order to speed up independent parts of the simulation such as random roll-outs which are mutually exclusive of each other. All the AI implementations make use of game supervisor classes.

1) Random Agents: I used Marsaglia's xorshift generator [13] [14], the well known Fast-Rand PRNG [11] and the Mersenne Twister 19937 generator (which is a built into c++11, and recommended to be used) as the PRNGs for my random agents [2]. These PRNGs are used to generate a random number which is in the range of the number of cards in that player's (agent's) hand. The agent then returns the corresponding card to be played by the framework.

2) Heuristic Agent: The heuristic player uses the equation 1 where the reward is between -1 and 1. This Heuristic takes into consideration the current attack-points each player has in play for that round, the number of cards left in each player's hands and the number of round wins - which is weighted more heavily. The reason the terms are normalized is so that their influence with regards to the reward is weighted according to the observed importance of each term within the game.

$$\begin{aligned} reward = 0.2 \text{ X (AttackPoints - EnemyAttackPoints)}+ \\ 0.2 \text{ X (HandSize - EnemyHandSize)}+ \quad (1) \\ 0.6 \text{ X (RoundWins - EnemyRoundWins)} \end{aligned}$$

3) Random Roll-out Agent: The random roll-out agent iterates through its own hand (deck). For each card it simulates an entire random game (to the end of the game) and will repeat the simulations based on the limit defined by the variable numberOfIterations. This limit is set to 1000 iterations for most of the experimental runs. For each simulation it records the wins, draws and losses and when the agent finishes it will return the move (Card) with the most wins. This agent uses OpenMP to parallelise the simulations run for a given card.

4) MCTS Agent: This agent follows the MCTS framework as defined by Baier et al. [3]. The selection phase will initially select randomly (using the Mersenne Twister approach) and as the tree begins to develop it will select based on the back-propagated UCT values for each node (in the tree). This phase uses the default UCB1 sampling algorithm. The cumulative reward which is used during the back-propagation phase is a decimal number between -1 and 1. If there is a win for the agent for a specific roll-out then 1.0 is added to the specific node being computed. If there is a draw than 0.5 is awarded, otherwise if there is a loss than -1 is added to that node. The back-propagation step calculates the value of each node by dividing the cumulative reward at that node by the number of times the node has been visited. This agent also has a limiting parameter numberOfIterations which is set to 1000 iterations for most of the experimental runs.

5) MCTS2 Agent: This agent modified the basic MCTS framework so that pruning of the tree takes place at every iteration of the algorithm. Rather than using the BMCTS algorithm to prune the tree based on a heuristic, this agent deletes the worst paths which have been found. This is determined during the back-propagation phase where the the best at that specific iteration is found and worse paths are left in the tree.

IV. Results and Discussion

A. Random Agents

The random agents play an important role in the investigation of Gwent as they are able to show the effect a strategy (or lack thereof) has on the overall outcome of a game. As shown in table (I), the MT agent shows a good spread of results over wins, losses and draws when it plays itself. The XorShift also shows to have this property. This is because they are fairly evenly spread PRNGs and so the expectation is that results are evenly spread over all outcomes in the domain as long as they are all equally likely. It is interesting to note that the Fast Rand agent appears to be biased when playing itself as it is only able to draw with itself. When it plays any of the other two random agents it loses badly - The opposing agent wins around 98% of the time in both cases. Fast Rand is a poor PRNG which has displayed a bias in terms of the numbers it generates [11]. It is seeded with entropy so the bias observed has to be caused by the algorithm itself.

TABLE I. TABLE OF EMPIRICAL RESULTS FOR PURELY RANDOM PLAYS (OVER 10000 GAMES, N=1000)

Agents (P1 vs P2)	P1 Wins (%)	P2 Wins (%)	Draws (%)
MT vs MT	34.89	36.40	28.71
MT vs Fast Rand	98.23	0.11	1.66
MT vs XorShift	35.60	35.52	28.88
Fast Rand vs Fast Rand	0.0	0.0	100.00
Fast Rand vs XorShift	0.16	98.11	1.73
XorShift vs XorShift	34.99	35.92	29.09

Note: In table (I) that the Fast-Rand performs the worst of any agent whilst MT and Xorshift are evenly matched.

The table (II) shows that fast rand is by far the fastest agent being considered. This speed comes at the cost of making reliable and intelligent decisions.

TABLE II. TABLE OF AVERAGE DECISION TIMES FOR RANDOM AND HEURISTIC AGENTS

Agent	Average Time
MT	0.000646468
FAST Rand	0.000238346
Xorshf	0.000557472
Heuristic	0.0033047

Note: In table (II) that the Fast Rand is the fastest agent whilst the heuristic is the slowest.

B. Heuristic Agent

In the table (III), the heuristic agent appears to produce the same results as the random agents it is pitted against. This could be because the parameters of the reward function being used needs to be adjusted (and trained) or that the reward function is not adequate to make informed decisions about the game. Since the hands are randomised the results seem to reflect that the heuristic has the same chance of winning a game (and choose the ”right” moves) as a random agent. This means that the heuristic is no better than a random agent. A solution would be to build in a set of logical states which encode a strategy to improve the heuristic agent.

TABLE III. TABLE OF EMPIRICAL RESULTS FOR DECISION MAKING AGENTS VERSUS THE RANDOM AGENTS (OVER 1000 GAMES, N=1000)

Agents (P1 vs P2)	P1 Wins (%)	P2 Wins (%)	Draws (%)
Heuristic vs MT	31.82	34.22	33.96
Heuristic vs Fast Rand	99.34	0.48	0.18
Heuristic vs XorShift	31.97	34.67	33.36
Rollout vs MT	51.875	28.125	20.0
Rollout vs Fast Rand	97.5	0.0	2.5
Rollout vs XorShift	55	28.125	16.875
MCTS vs MT	61.82	25.45	12.72
MCTS vs Fast Rand	99.09	0.0	0.91
MCTS vs XorShift	60.45	25.0	14.55
MCTS2 vs MT	49.5	32	18.5
MCTS2 vs Fast Rand	99.0	0	1.0
MCTS2 vs XorShift	54	25.0	21.0

When the AI agents such as MCTS and random roll-out are played against the random agents (With parameter values $N = 1000$ for the agents) the AI agents dominate the game and have considerably higher win rates.

C. Random Roll-out Agent

The Random roll-out agent (at $N = 1000$) performs very strongly against all three random agents in table (VI). This shows that having some information about the game state (the effectiveness of a card under a certain state) works well and that a strategy will effect the outcome of a game of Gwent. However the number of iterations directly effects the choice the roll-out agent makes. If the value is too small the roll-out agent is as effective as a random agent but much slower. However a large number of iterations would make the agent impractically slow as shown in table (IV). There is also diminishing returns as seen in figure IV-C, where the upwards trend of winning and the downwards trend of drawing and losing games of Gwent are very shallow. This means that sacrificing time and computationally complexity will not necessarily make the algorithm able to win the game every time. This can be explained by the stochastic nature of the game and also that the random roll-out agent is limited in that it cannot truly strategise real opponent moves - its is reliant on random moves with a theoretical upper bound on the accuracy (in terms of good move choices) it can attain.

TABLE IV. TABLE OF AVERAGE DECISION TIMES FOR DECISION AGENTS AT INTERVALS OF N

Agent	N=10 (secs)	N=100 (Secs)	N=1000 (Secs)
Rollout	0.111459	1.47849	108.055421519
MCTS	0.0920657	1.52643	103.5874541284
MCTS2	0.0894733	1.50069	92.4551747475

Note: In table (IV) that the MCTS2 is the fastest agent whilst the Random Roll-out is the slowest agent.

D. MCTS Agents

From the table (VI) the standard MCTS implementation (MCTS) can be seen to work exceptionally well against the random agents. Even at a small number of iterations ($N = 10$) it can be seen in the table (V) that the agent is able to win more than the random agents. It is however evenly matched with the random roll-out agent in this configuration. This is because at 10 iterations the algorithm will barely cover all the possible moves available in a given round (i.e. cards in the players hand) so it will effectively be doing random roll-outs and will not build a very deep tree. As N increases however, MCTS becomes a far better agent than the random roll-outs one. MCTS begins to win considerably more games over roll-out with far fewer draws. At $N = 1000$ MCTS is the most successful agent in comparision to all of its opponents

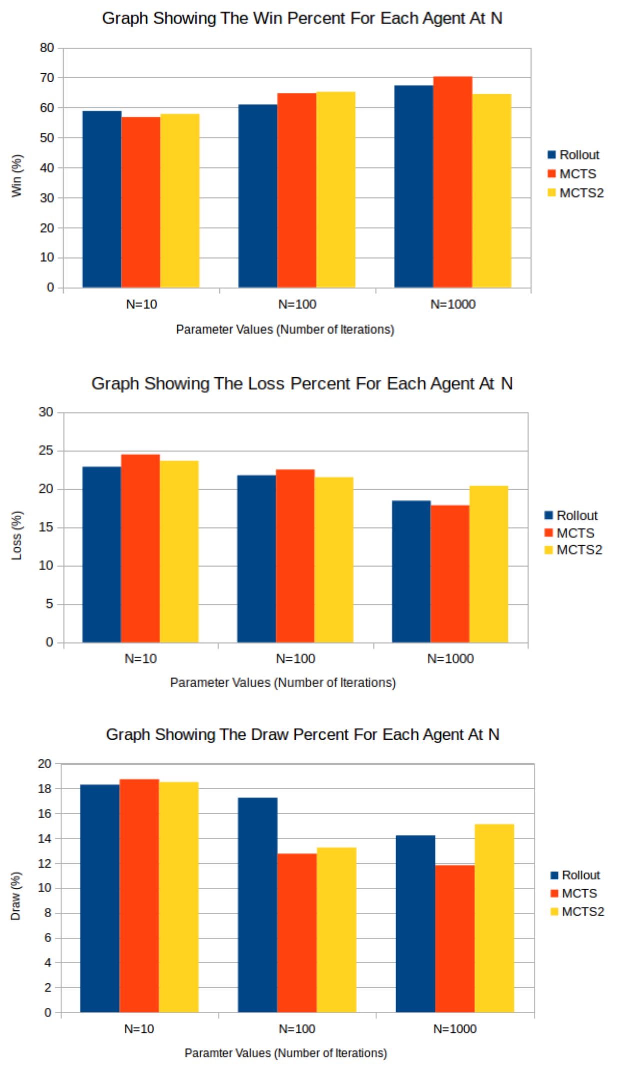

Fig. 5. Comparison of Rates of Decision Agents Against An Accumulation Of All The Random Agents Results. Note: A higher the win percent is better but a lower percent is better for the other graphs.

as seen in table (VI) and table (III).

TABLE V. TABLE OF EMPIRICAL RESULTS FOR DECISION MAKING AGENTS PLAYED AGAINST EACH OTHER (OVER 1000 GAMES), N IS THE NUMBER OF ITERATIONS

Agents (P1 vs P2)	N	P1 Wins(%)	P2 Wins(%)	Draws(%)
Heuristic vs Rollout	10	33	36	31
Heuristic vs MCTS	10	21	40	39
Heuristic vs MCTS2	10	26	42	32
Heuristic vs Rollout	100	34	36	30
Heuristic vs Rollout	1000	35	33	32
Heuristic vs MCTS	100	28	54	18
Heuristic vs MCTS	1000	16	63	21
Heuristic vs MCTS2	100	27	54	19
Rollout vs MCTS	10	35	33	32
Rollout vs MCTS2	10	29	39	32
Rollout vs MCTS	100	26	58	16
Rollout vs MCTS	1000	26	51	23
Rollout vs MCTS2	100	16	64	20
MCTS vs MCTS2	10	46	40	14
MCTS vs MCTS2	100	51	39	10

The second MCTS agent (MCTS2) appears to produce very similar results to the original algorithm (In table V). It is weaker and wins less games than the vanilla algorithm but is much faster (on average decisions) than the vanilla algorithm as seen in figure (6) and table (VI). This is because the search space in the generated tree is greatly reduced. If the loss of accuracy in decision making is acceptable for the increase in decision speed than this algorithm would be a good choice. The random rollout agent does become a better at the game when the number of iterations increases. This is because MCTS2 does not take into account all the possible actions available whereas the random rollout agent (given enough iterations) will consider all actions multiple times.

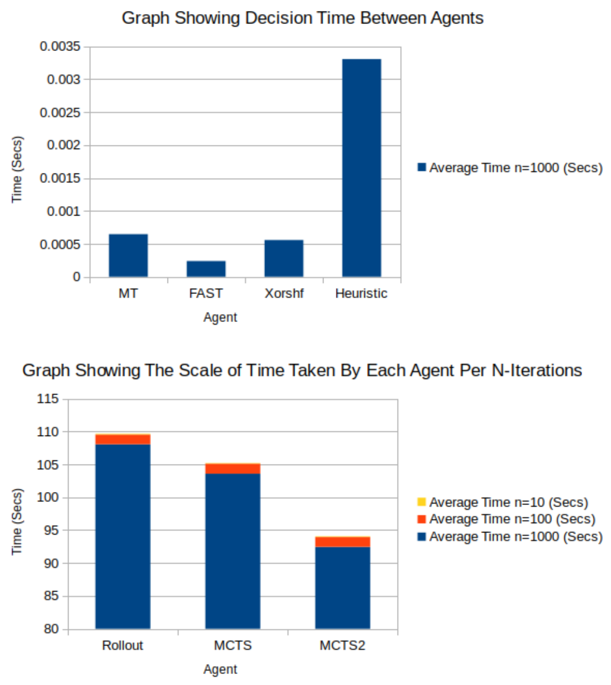

Fig. 6. Average Decision Time of Agents

Note: In figure (6) that the average time for $n = 10$ is so small in comparison to the other values for n that it does not register on the graph.

TABLE VI. Table of Empirical Results For Decision Making Agents Versus The Random Agents (Over 1000 games, N=1000)

Agents (P1 vs P2)	**P1 Wins (%)**	**P2 Wins (%)**	**Draws (%)**
Heuristic vs MT	31.82	34.22	33.96
Heuristic vs Fast Rand	99.34	0.48	0.18
Heuristic vs XorShift	31.97	34.67	33.36
Rollout vs MT	51.875	28.125	20.0
Rollout vs Fast	97.5	0.0	2.5
Rollout vs XorShift	55	28.125	16.875
MCTS vs MT	61.82	25.45	12.72
MCTS vs Fast	99.09	0.0	0.91
MCTS vs XorShift	60.45	25.0	14.55
MCTS2 vs MT	49.5	32	18.5
MCTS2 vs Fast	99.0	0	1.0
MCTS2 vs XorShift	54	25.0	21.0

V. Conclusion and Future Research

These AI agents have shown that the game of Gwent is not solely determined by chance but that strategy also plays a role in the outcome of a game. The Random agents have been beaten by the agents who explore thier action space and the resultant outcomes of those actions. The heuristic AI did not perform as expected which shows that whilst the game is not random, a heuristic which relies on a linear combination of facts from the current game state is too brittle an AI to make informed decisions within the game. The brute force approach of simulating all possible moves in the game has benefits and will produce usable decision but it is a limited strategy which does not work well in very large domains such as Gwent. MCTS helps to solve this problem by limiting its search space in the regions of the state-space where the decisions are promising. This means that MCTS is able to find good decisions often but requires lots of memory and computational time. MCTS is however also used as a base framework for far more complex algorithms which apply other AI and algorithmic techniques to improve the perforamce of such tree search algorithms. MCTS was also shown to be the most successful agent when run against all the other agents that were implemented.

Gwent is a complex and interesting domain which has provided a challenge which incorporates stochasticity and non-linear decision making to the problem posed to the AI agents. This project was limited in scope. Future work should look at opening up the domain to more complex rule sets and adding in extra hidden information such as making the opposing decks unknown. Another interesting avenue would be to have the oppenents know each other's hands but not their own hands. The heuristic should also be improved by using more paramters which are trained over random games and the implementation of logic based strategies should also be investigated as a possible heuristic AI. Research into random sampling techniques and machine learning to assist MCTS in sampling good moves is another anvenue of interest.

Acknowledgements

I would like to acknowledge Dr. Richard Klein for his help and support with debugging the implementation of the testing framework used in this project. I would also like to thank Mr Steven James for his help and support in our discussions about tree search agents and MCTS-based algorithms.

Finally I want to state that all intellectual property associated with the game Gwent (such as images of cards) is owned by CD PROJEKT RED and this property has been used according to the Gwent Fan Content Guidelines for an unofficial fan work [1].

Glossary of Terms

Abbreviation	**Meaning**
AB	Alpha-Beta
AI	Artificial Intelligence
ACML	Adaptive Computation and Machine Learning
BMCTS	Beam Monte-Carlo Tree Search
DIP	Digital Image Processing
HPC	High Performance Computing
ISMCTS	Information Set Monte-Carlo Tree Search
MCS	Monte-Carlo Simulation
MCTS	Monte-Carlo Tree Search
MCTS-MB	MCTS with Minimax Backpropagation
MCTS-MR	MCTS with Minimax Rollouts
MCTS-MS	MCTS with Minimax Selection
MDP	Markov Decision Processes
MT	Mersenne Twister
PRNG	Pseudo-Random Number Generator
UCB	Upper Confidence Bounds
UCT	Upper Confidence Bounds for Trees
XorShft	Xorshift

References

[1] Fan Content Guidelines for Gwent. ”https://www.playgwent.com/en/fan-content”.

[2] Mersenne twister 19937 generator, c++ documentation. ”http://www.cplusplus.com/reference/random/mt19937/”.

[3] H. Baier and M. H. M. Winands. Beam monte-carlo tree search. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 227–233, Sept 2012.

[4] H. Baier and M. H. M. Winands. Monte-Carlo Tree Search and Minimax Hybrids. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, Aug 2013.

[5] Bronson, G.J. *C++ Programming: Principles and Practices for Scientists and Engineers*. Cengage Learning, 2012.

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[7] CD PROJEKT S.A., CD PROJEKT RED. GWENT Guide. pages 1–4. Poland, 2015.

[8] Chaslot, Guillaume and Bakkes, Sander and Szita, Istvan and Spronck, Pieter. Monte-Carlo Tree Search: A New Framework for Game AI. In Christian Darken and Michael Mateas, editors, *AIIDE*. The AAAI Press, 2008.

[9] Chaslot, Guillaume M. J-B. and Winands, Mark H. M. and van Den Herik, H. Jaap and Uiterwijk, Jos W. H. M. and Bouzy, Bruno. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation (NMNC)*, 04(03):343–357, 2008.

[10] D.J. Edwards and T.P. Hart. The Alpha-Beta Heuristic. Technical Report AIM-30, Massachusetts Institute of Technology, http://hdl.handle.net/1721.1/6098, 1961.

[11] Christopher Owens (Intel). Fast random number generator on the intel pentium 4 processor. "url=https://software.intel.com/en-us/articles/fast-random-number-generator-on-the-intel-pentiumr-4-processor/", Jun 2017.

[12] Kocsis, Levente and Szepesvári, Csaba. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293. Springer-Verlag, 2006.

[13] George Marsaglia. Random numbers for c: End, at last? "url=http://www.cse.yorku.ca/ oz/marsaglia-rng.html".

[14] George Marsaglia. Xorshift rngs. *Journal of Statistical Software, Articles*, 8(14):1–6, 2003.

[15] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach (3rd Edition). chapter 5, pages "161–189". Pearson, https://www.amazon.com/gp/product/0136042597, 2009.

[16] S. James and G.D. Konidaris, and B. Rosman. An Analysis of Monte Carlo Tree Search. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[17] Victor Eijkhout with Robert van de Geijn and Edmond Chow. *Introduction to High Performance Scientific Computing*. lulu.com, 2011. http://www.tacc.utexas.edu/~eijkhout/istc/istc.html.

[18] Daniel Whitehouse, Peter I. Cowling, Edward Jack Powley, and Jeff Rollason. Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-13, Boston, Massachusetts, USA, October 14-18, 2013*, http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/view/7369, 2013.