Developing an Agent to Play the Card Game Cheat

Paul Sinclair

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2018

Abstract

Monte Carlo Tree Search (MCTS) has seen incredible success in recent years in a number of domains, most notably the ancient strategy game Go [1]. In this paper, I investigate the potential of applying similar techniques to a new problem: the card game Cheat. I develop a number of different stochastic rule-based agents to demonstrate the challenges involved in this domain, and implement a new agent that uses Perfect Information MCTS, a variant of standard MCTS which supports games of imperfect information. I explore the challenges and benefits of using PIMCTS in this domain, and compare the performance of the PIMCTS agent to the rule-based agents.

Acknowledgements

Many thanks to Alex Lascarides, my supervisor, for all of her wonderful advice and support throughout my project, and to Mihai Dobre both for providing his MCTS framework and generally for all of his assistance.

Thanks to Connie Crowe, Will Mathewson and Šimon Rovder for proofreading my report and providing useful suggestions for improvements. Thanks also to my parents for their support and advice, and to all my friends at Edinburgh University for helping me through all the long nights I spent in the computer labs.

Table of Contents

Chapter 1

Introduction

Intelligent agent development has recently seen some incredible success across several different domains. Arguably the most notable of these is AlphaGo's 4-1 victory against the world champion Go player Lee Sedol [2], an achievement previously estimated to be at least 10 years away [3]. AlphaGo's success is due to two key features: a pair of deep neural networks which evaluate game positions and select moves, and a version of Monte Carlo Tree Search (MCTS) which combines Monte Carlo simulations with the output of these networks [1]. Other agents based on MCTS have also produced world class results in games including Hex [4], Havannah [5], and Sudoku [6].

However, most of the work on MCTS has focussed on deterministic games of perfect information, i.e. games without any element of chance, where the full game state is public [7]. In particular, though a number of possible ways of extending MCTS to support nondeterministic games have been devised, finding an effective system either in general or even for a specific domain is still very much an active research topic. Here, I explore the effectiveness of MCTS in one such nondeterministic game: the card game Cheat. I develop a number of rule-based stochastic agents to demonstrate the challenges involved in the game, showing some strategies which help overcome them. I then implement an agent which uses Perfect Information MCTS (PIMCTS) to decide which actions to take and evaluate this against the rule-based agents, discussing the strengths and weaknesses of PIMCTS in this domain.

1.1 Cheat

Cheat [8], also known as Bullshit, I Doubt It, and Liar [9], is a well known card game which features a large element of deception. The basic rules are as follows:

- All 52 cards of a standard deck of cards are dealt out evenly to all players.
- Play proceeds in a clockwise direction, starting from the player on the left of the dealer.
- On each player's turn, they play one to four cards face down in the discard pile,

and make a claim as to those cards' value, e.g. "I just played three Jacks". Typically, there is some restriction on which values may be claimed. All other players may then challenge this claim if they do not believe it.

- If no players challenged, the next player takes their turn.
- If another player did challenge, the original player's cards are revealed. If they were indeed cheating, the original player must take all the cards from the discard pile, then it becomes the turn of the player who challenged. If however they were telling the truth, the player who challenged must take all the cards from the discard pile, and the original player may play again.
- Play continues until one player gets rid of all their cards: this player is then the winner![1]

Cheat has the following properties:

1. Partially observable: Players cannot see other players' hands or played cards.
2. Stochastic: Chance dictates the initial deal of the cards, though from the dealt state onwards it is deterministic.
3. Simultaneous: Many players have the opportunity to challenge at the same time.
4. Infinite: There exists a sequence of moves such that the game never ends. For example: Player A plays one Ace honestly. Player B challenges, so has to pick up the Ace, and Player A gets to play again. Player A plays one King dishonestly. Player B challenges, so Player A picks up the card they just played, and it becomes player B's turn. Repeat with the players swapped.
5. Multi-agent: Several players are competing to win.
6. Known: All the rules defining the game are public.
7. Discrete: Time can be split into turns.
8. Static: The state does not change without any of the agents acting.

These features, in particular the lack of perfect information due to chance and partial observability, suggest that developing a good agent to play Cheat will be an interesting challenge. Cheat also shares many more of its properties with a typical real world scenario than most games which MCTS has been applied to, so any research in this area has the potential to be applicable to a much wider range of useful problems.

1.2 Project Aim

There were two main aims for the project:

1. To develop an agent which plays Cheat as well as possible.

[1] Note that if you play your last card(s), but then someone correctly challenges you, you must pick up all the cards from the discard pile, and so have not yet won.

2. To investigate how feasible MCTS-based techniques are for tackling Cheat, as there has been no research published in the open literature on using these approaches in this domain.

In Chapter 2, I go over some related work, investigating others' attempts and approaches to developing an agent to play Cheat, and other results from using MCTS in similar domains. In Chapter 3, I discuss some of the design and implementation decisions that I made when formalising the problem. In Chapter 4, I describe the stochastic rule-based agents I developed and evaluate their performance. In Chapter 5, I discuss the MCTS approach, the challenges but also the potential that MCTS offers in this domain, the PIMCTS agent I developed, and its performance.

My contributions:

- Designing and implementing the Cheat environment and agent comparison system.
- Designing and implementing seven different types of stochastic rule-based agents for Cheat and evaluating their performance.
- Integrating the Cheat MDP into an existing standard MCTS system and adapting it to run PIMCTS.
- Integrating opponent modelling into the PIMCTS agent.
- Evaluating the PIMCTS agent and discussing the benefits and challenges of using PIMCTS in this new domain.

Chapter 2

Related Work

2.1 Other Cheat Agents

There have been a number of previous attempts to develop agents to play Cheat.

Philip [10] describes her implementation of three simple stochastic rule-based agents for Cheat. All use a very basic pure strategy without utilising a game model or opponent model. None seem to play at a particularly high standard, and would likely be beaten by any reasonable human or agent, though we cannot test this as she has not released her source code.

Hacohen and Doron [11] attempt to develop an agent that uses Q-learning [12] to develop a strategy for a greatly simplified version of Cheat, where each agent plays only one card at a time, claiming both the value and the suit. They develop three very basic rule-based agents, then train and evaluate a learning agent against each of these in turn. This learning agent models the current state based on seven basic features, then learns which is the best action to take from each of these states using Q-learning. They found that this agent could learn a good policy against the agent which regularly cheated, but was unable to learn a strategy to even equal either of the agents which never choose to lie, winning less than 5% of games against both. I believe their main mistake was their choice of state representation; there is not enough entropy within seven features to sufficiently model all the information about the state of the game that is necessary to choose the best action. Their method has the benefit of adapting its strategy based on the opponent's actions, but this comes with the downside of needing to be retrained for each new type of agent it plays against.

Tamari and Peterfreund [13] also attempt to develop a Q-learning-based agent for Cheat. They develop a much larger and more robust set of features to represent each state, successfully capturing much more of the useful information than Hacohen and Doron. When choosing what cards to play, Tamari and Peterfreund enforce a number of restrictions on possible moves that the agent may take in order to reduce the search space, then the agent picks which of these to take based on a policy learned using Q-learning. For deciding whether to challenge, the agent first applies a number of hard-coded rules (such as challenging if they are able deduce that the other

player must be lying based on the cards currently in their hand), then if none of these rules gives a definite answer, it uses a simple dynamic opponent model [14] to predict whether this player chose to cheat. This agent fared fairly well against most of their basic rule-based stochastic agents in a 1 v 3 match, but its performance suffered significantly when playing against multiple different types of agents. Overall, I think this approach shows good promise for learning a policy for playing, but like both Philip's [10] and Hacohen and Doron's agents [11], it loses lots of important information by not maintaining a belief state for the game.

All in all, it seems the current state-of-the-art Cheat agent is fairly simplistic. An agent which models a belief state about the hidden information in the world and uses this appropriately is likely to outperform many if not all of the previous agents developed. It also appears that no one has yet tried using MCTS-based techniques to tackle Cheat, so any research into the applicability of these techniques to this domain would prove insightful.

2.2 MCTS Agents in Related Domains

While it appears no one has yet tried using MCTS-based techniques for Cheat, they have proved successful in a number of similar domains.

Poker shares a number of features with Cheat: it is a card game of imperfect information where a key part of the game is bluffing and predicting when your opponent is bluffing. Van den Broeck et al. [15] show that standard MCTS can perform well in this domain after adapting the search tree to replace all hidden information and opponents' actions with chance nodes. Ponsen et al. [16] build upon this by integrating an opponent model with MCTS, improving the accuracy of the probability distribution at each chance node.

Another similar game for which MCTS-based techniques have proved successful is the Chinese card game Dou Di Zhu [17]. Whitehouse et al. [17] compare two different MCTS-based approaches for Dou Di Zhu, Perfect Information MCTS (PIMCTS) and Information Set MCTS (ISMCTS), and conclude that while both are reasonably effective in this domain, ISMCTS performs better due to avoiding some of the problems which arise when "averaging over clairvoyance" [18]. On the other hand, PIMCTS performs very well in the game of Skat [19], performing roughly at the level of human experts.

Overall, it seems plausible that MCTS-based techniques will have a good chance of performing well for Cheat due to their success in many other similar card games.

Chapter 3

Designing the Game Environment

3.1 Designing the Game

3.1.1 Selecting Among Rule Variations

Like many card games, the game of Cheat has passed from person to person by word of mouth over many years. As a result, there are many different versions of Cheat with slightly different rules, so the first thing I needed to do was pick which variants to use.

Note that there are also other games which go by a similar name but have significantly different mechanics, for example including a central deck to draw more cards from. In this paper I consider those as totally separate games, not different versions of the Cheat game discussed here, so I will not include them in the discussion.

3.1.1.1 Legal Values to Claim

In most versions of Cheat, the player is restricted in which values they can claim when playing cards, usually to the values directly above, below, or equal to the previous value, or some subset of these. Using no restrictions means players are never obliged to cheat, which can result in very dull games with boring strategies, while too strict restrictions cause such a large majority of moves to be cheats that each round is too short to be interesting. Therefore, I chose to restrict it to any of the values directly above, below, or equal to the previous value, as I felt that this option allowed for a good balance between being able to play legally and not.

Similarly, when the discard pile is empty, most versions take this to be a 'reset', allowing the next player to claim any value, while some keep the restrictions that would have been there had the previous move not been challenged. I chose the former option, as this has the potential to allow tactics to develop based around which value to start off the new round with, one of the more interesting decisions in the game.

3.1.1.2 Minimum Number of Cards

Different versions of Cheat set the minimum number of cards that must be played each turn from zero (allowing passes) to two. I chose to set the minimum to one for a similar reason to above, in order to produce a good balance between being able to play legally and not.

3.1.1.3 Player Turn After Challenge

Another rule that varies greatly is whose turn it is after a challenge. The most common three are A) whoever picked up, B) whoever did not pick up, or C) no turn order change after a challenge; the player who would have gone next starts. I chose option B because it provides extra incentive for challenging a dubious claim yourself rather than letting someone else take the risk.

3.1.1.4 Discard Reveal After Challenge

Another variation lies in publicly revealing all, some, or none of the discard pile after a challenge. In most game between humans, this choice makes little difference, as there is no chance of players remembering all the cards from the discard pile and using this information, but an AI has no such problems with remembering many details. I decided to not reveal any information about the discard pile after a challenge, other than to the player who picks it up of course, as otherwise a good AI would fairly swiftly learn exactly what each player's hand was at the start of each round.

3.1.1.5 Jokers

Some people choose to add Jokers to the deck as 'wild cards', therefore also increasing the maximum number of cards that can be played each turn. I decided not to include Jokers for two reasons. Firstly, it allows deductions for which moves must be illegal to have a much higher recall at 100% precision. Secondly, this prevents the strategy of keeping a Joker until your final card and going out on that without being able to be challenged; though a good strategy, it effectively just gives the players who are dealt the Jokers a large advantage without otherwise significantly affecting gameplay.

3.1.2 Constraints

Historically, Cheat is designed to be played face to face with a physical deck of cards, so I had to make a number of small changes in order to create a version which can be played both on and by a computer.

3.1.2.1 No Lying About Number of Cards Played

When playing the game with a physical deck of cards, players are allowed to lie about the number of cards they played as well as their value, for example by carefully hiding cards behind one another. However, this does not translate well to a digital version since there is no way of concealing the number of cards you played. It also adds little to the strategy of the game; hand sizes are always public, so all that is required to beat this strategy is to always challenge if the opponent has not played the claimed number of cards.

3.1.2.2 Sequential challenging

Normally, after a player has taken their turn, any other player can challenge at any point until the next player makes their move. This is often also a point of key psychological warfare between players as they wait to see who, if anyone, will challenge a potentially dubious claim. In my implementation however, I chose to remove this feature, instead giving each player in turn a single chance to challenge.

This simplifies the environment by removing its only simultaneous element, ensuring that only one player has the opportunity to challenge at each time, and making it fairer when faster agents play against slower agents (such as humans) as both players can be given the time they require to reach a decision. It also speeds up the simulation by preventing waiting from being a valid action.

3.1.2.3 No Psychological Factors

When played face to face, Cheat is primarily a social deduction game: one of the main challenges is to figure out whether another player is lying or not. Many key factors which players use to deduce if other players are lying are based on group psychology, for example body language, confidence, reactions to game events, and speed of play. Most of these are clearly impossible to emulate when playing digitally rather than face to face; time taken to play is perhaps the only detectable psychological factor.

This does change the style of gameplay quite significantly, as it becomes much more about the logical deductions about the cards than the human judgement of the honesty of an individual. In the end I opted not to take even detectable psychological factors like timing into account, largely because most of my work involves comparing two agents rather than an agent and a human, so there is little or no psychology to even try to deduce.

3.1.3 Final Rules

Taking all of this into account, the final rules that I settled on were as follows.

- All 52 cards of a standard deck of cards are dealt out evenly to all players.

- Play proceeds in a clockwise direction.
- On each player's turn, they play one to four cards from their hand to the discard pile, without revealing which cards they played, making a claim as to those cards' value, e.g. "Three Jacks". The value claimed must be within a single step of the previous value (e.g. you can claim ten, Jack, or Queen if the previous value was a Jack). If the discard pile is currently empty, any value can be claimed.
- One by one, going clockwise, the other players get the opportunity to challenge this claim.
- A challenge is correct if any of the cards played were not of the value claimed.
- If no players challenged, the next player takes their turn.
- If a player challenges correctly, the original player takes all the cards from the discard pile, then it becomes the turn of the player who challenged. If a player challenges incorrectly, the player who challenged must take all the cards from the discard pile, and the original player may play again.
- Play continues until one player gets rid of all their cards: this player is then the winner. If a players plays their last card(s), but then someone correctly challenges them, they must pick up all the cards from the discard pile, and so have not yet won.

3.2 Agent Constraints

For reasons of time and complexity, there were a number of constraints I fixed for all agents I designed.

3.2.1 Number of Players

It is possible to play Cheat with two or more players, with no real maximum. With only two players, a number of cards are randomly discarded in order to improve the initial uncertainty. If playing with more than eight players, extra decks are used, with a rate of one for every eight players in the game (i.e. number of decks $= \lceil \frac{\text{number of players}}{8} \rceil$). Despite these options existing, the recommended number of players is usually 3-6, as too many players can cause the game to drag on for a very long time.

For the sake of simplicity, I decided to fix the number of players to a single value, rather than trying to develop an agent which has a good strategy for all different numbers of players. This also meant that implementing the game was easier as various potential variables were fixed, including in particular the number of cards in the game. The number of players I chose was four, as four player games tend to have the best balance between players.

3.2.2 Adaptive Behaviour with Dynamic Opponent Modelling

When humans play Cheat, one of the common strategies is to try and work out other players' behaviour, such as how frequently they choose to cheat and how confident they are before they challenge (Opponent Modelling), then to modify their own behaviour based on this (Adaptive Behaviour). Implenting these tactics in artificial agents is an active research area in AI development [14], both in order to improve an agent's win rate [20], and to try to more closely match the other players' level in order to provide a more entertaining gaming experience for the user [21].

Despite this being an important part of a strong strategy, I decided that both dynamic opponent modelling and adaptive behaviour were out of scope for this project. This makes modelling the game state much easier, as you do not need to keep track of each players' behaviour over the course of the game(s).

I do however use static opponent modelling, which makes assumptions about how a typical agent is likely to play, but then does not update this based on observed actions. This helps improve the agent's beliefs about the world (and therefore quality of play) against typical agents, but also allows a learning agent (such as a human) to behave contrary to these assumptions to deliberately throw off the agent.

3.2.3 Time/Space Constraints

As we will discuss in Section 5.1, it is theoretically possible for any agent to produce an optimal strategy with sufficient time and space. However, this is clearly entirely impractical. Instead, I set some rough bounds on the maximum time and space available for each agent to take their turn. I aimed for a one second maximum time for each agent's decision, using no more space used than is available on a typical modern PC. I did not strictly enforce these bounds, instead I chose appropriate meta-parameters (for example, number of MCTS iterations) where necessary in order to roughly stick within these limits.

3.3 Comparing Agents

In order to compare two different agents to see which is better, I simulate many games with these two agents and see which has the higher win rate. However, there are a number of different factors which must be taken into account.

3.3.1 Player Counts

Because I have chosen to work in a world with four players, we cannot simply compare two agents by playing games of AGENT 1 against AGENT 2, as there need to be four agents playing in total. Instead, we will run each test twice: once through with one

AGENT 1 playing three AGENT 2s, and once through with one AGENT 2 playing three AGENT 1s.

An alternative would be to have two 'dummy' agents play along with one AGENT 1 and one AGENT 2. However, choosing what sort of agent to use as these dummy agents and implementing this agent would then be its own separate challenge, and would also introduce more bias into the results, as depending on how this dummy agent played, the two agents being compared could fare very differently. As such, I chose the former option for this work.

3.3.2 Accounting for Player Position Bias

Clearly not all players are equal at the start of a game of Cheat, as one player gets to play first and so get rid of cards earlier. Exactly how much of an advantage or disadvantage playing first is will be different for different agents, but it will always be a variable which we want to eliminate when comparing two agents. As such, when running tests, the player order will change each game such that each possible player order will occur an identical number of times.

3.3.3 Ensuring Significant Results

I use a Pearson chi-squared test [22] in order to help ensure any results observed when comparing two agents are significant and not simply due to random chance. Here, the null hypothesis is that both agents have an equal chance of winning, so in particular that when playing N games between one AGENT 1 and three AGENT 2s, AGENT 1 will win $\frac{N}{4}$ games. Using Pearson's formula, we have that

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{3.1}$$

$$= \frac{(O_{wins} - E_{wins})^2}{E_{wins}} + \frac{(O_{losses} - E_{losses})^2}{E_{losses}} \tag{3.2}$$

$$= \frac{\left(O_{wins} - \frac{N}{4}\right)^2}{\frac{N}{4}} + \frac{\left(O_{losses} - \frac{3N}{4}\right)^2}{\frac{3N}{4}} \tag{3.3}$$

$$= \frac{12\left(O_{wins} - \frac{N}{4}\right)^2 + 4\left(O_{losses} - \frac{3N}{4}\right)^2}{3N} \tag{3.4}$$

where O_{wins} is the number of wins observed, O_{losses} is the number of losses observed, and N is the total number of games played. Because $O_{wins} + O_{losses} = N$, we can rewrite

		Significance level				
		0.900	0.950	0.975	0.990	0.999
N	40	15	16	17	18	20
	100	33	34	35	37	40
	500	141	144	147	150	157
	1000	273	277	281	286	296

Table 3.1: Number of wins needed out of N games for each significance level

this as:

$$\chi^2 = \frac{12\left(O_{wins} - \frac{N}{4}\right)^2 + 4\left((N - O_{wins}) - \frac{3N}{4}\right)^2}{3N} \tag{3.5}$$

$$= \frac{12\left(O_{wins} - \frac{N}{4}\right)^2 + 4\left(O_{wins} - \frac{N}{4}\right)^2}{3N} \tag{3.6}$$

$$= \frac{16\left(O_{wins} - \frac{N}{4}\right)^2}{3N} \tag{3.7}$$

Rearranging for O_{wins} gives us:

$$O_{wins} = \frac{N + \sqrt{3N\chi^2}}{4} \tag{3.8}$$

This gives us an easy formula to use to calculate how many wins we need to ensure a statistically significant result over N games. In this problem, we have one degree of freedom (whether AGENT 1 wins or loses), so we read our values for χ^2 at each significance level from the first row of a χ^2 distribution table. Applying these values to the formula above, we can calculate a table of number of wins needed for various significance levels and number of tests, shown in Table 3.1. So whenever I compare two agents, I will report the statistical significance of this result based on this table.

3.3.4 No Global Best Agent

One point that is worth emphasising is that there is not necessarily one 'best' agent, i.e. one that will beat any other. Instead we could have a rock-paper-scissors scenario where each of three agents beats one agent and loses to the other. In fact, it is fairly easy to come up with some theoretical agents for Cheat where this would apply: AGENT 1 who both never challenges and never cheats (unless they have to), AGENT 2 who never challenges but always cheats, and AGENT 3 who always challenges and never cheats (unless they have to). AGENT 2 will beat AGENT 1 as by cheating he can get rid of his cards faster; AGENT 3 will beat AGENT 2 as by challenging, he forces AGENT 2 to keep on picking up the discards; and AGENT 1 will beat AGENT 3, as AGENT 3 will keep on challenging incorrectly and so picking up the discards.

As such, although we can compare two agents to see which is better against the other, or compare two agents to see which is better against some particular other agent, or on average against a number of different agents, we cannot say for certain that one agent is ‘best’.

Chapter 4

Creating the Stochastic Rule-Based Agents

The main aim of this stage of the project is to produce an agent that can play Cheat as well as possible, in particular better than an average or even expert human, but there are also a number of features I am aiming for.

First, I want to create an agent with a fully mixed policy. In other words, from any game state, the agent should have multiple actions with a non-zero probability of being chosen. The only exceptions to this should be in cases where there is a clearly optimal policy, such as challenging when a player has played their final card(s). This helps hide exploitable weaknesses which can otherwise be easily learned by a human or AI. For example, one early agent would always challenge if you played three cards on your first turn, so after only a couple of games against this AI, it was easy to adopt a strategy to exploit this.

Second, I want the agent to behave in a reasonably human-like manner. This is a very vague definition, but essentially I wanted games against the agent to feel like you were playing a good human, not just a computer. For example, an AI which appears to use a very basic strategy backed up with seemingly super-human memory would not qualify for this, while an AI playing what appears to be a strong strategy without purely relying on the extra processing power and memory of the computer would, even if they both win a similar number of matches against other agents.

Third, I want to produce an 'agent generator', a template which could produce infinitely many different agents based on a number of parameters. The initial parameters I wished to include were aggression (how likely they are to cheat when they have the option of playing honestly) and suspicion (how likely they are to challenge other players), but after further investigation more precise parameters were also included. This helps allow for easy comparison of different tactics, and also provides a much wider range of agents to test and evaluate against than would be possible by hard coding them all.

With these features in mind, I developed the following stochastic rule-based agents:

1. RANDOM: Baseline random strategy.
2. TIMID: Memoryless, as honest and trusting as possible.
3. NAIVE: TIMID with some basic memory.
4. NAIVE++: NAIVE with additional heuristics.
5. CUNNING$_p$: Cheats with probability p, but still as trusting as possible.
6. SUSPICIOUS$_{x,y}$: Maintains a probability distribution over the location of each card, challenges if the odds of the agent having those cards is below y.
7. BALANCED: Uses more advanced functions and heuristics when deciding whether to cheat and challenge.

Of these, RANDOM, CUNNING, SUSPICIOUS, and BALANCED are all 'agent generators' in that the behaviour of the resulting agent is determined by the inputted parameters, while RANDOM and BALANCED all have fully mixed policies, except in cases with a clearly optimal strategy.

4.1 RANDOM

The first agent I designed was a basic random agent. Though it performs very poorly and not at all like a human, it can nonetheless be used as some sort of a baseline for comparing other agents. On its turn, it will pick a single random card from its hand, then if the discard pile is currently empty (i.e. there are no restrictions on which value can be claimed), it will claim the true value, otherwise it will claim a value equal to the previously claimed value. When deciding whether to challenge, it will always challenge if that player would win otherwise. If not, it will challenge with probability p, a parameter which is set on initialisation. I will use RANDOM$_p$ to mean a random agent with a challenging probability of p.

Notably, even this simple agent design has allowed us to produce infinitely many different agents with different levels of suspicion (as defined above).

4.2 TIMID

The next agent I designed was a simple agent based on how I have seen beginners play Cheat. The TIMID agent will always play honestly if it can, playing all cards it has of that value. If it cannot play honestly, it will play one random card from its hand, picking a value to claim at random from the legal options. When deciding whether to challenge, it will always challenge if that player would win otherwise, otherwise it will only challenge if they have enough cards of that value in their hand to know the other player must be cheating (e.g. player claimed two Jacks, but they have three Jacks in their hand at the moment).

Comparing this agent to $\text{RANDOM}_{0.05}$, we see that this simple strategy is already a massive improvement, with TIMID winning approximately 49.6% of games against $\text{RANDOM}_{0.05}$ (detailed results can be seen in Figure 4.5).

4.3 NAIVE

The NAIVE agent is based on a similar principle to the TIMID agent, namely it plays honestly and trusts other players as much as possible, but this time it keeps a more detailed model of the world. For each card, it keeps track of all its possible positions (in each player's hand or the discard pile), then uses this model to challenge a player if it is impossible for them to have had that many cards of the claimed value in their hand. This effectively adds memory to the TIMID agent, though the uncertainty involved in which cards other players have discarded and/or picked up limits the effective length of this memory.

It is worth noting that modelling the possible position of each card in this way does not catch every situation where one could logically deduce that a move must be impossible. For example, if PLAYER 1 discards a card, then PLAYER 2 challenges incorrectly and picks up this card, the model naturally assumes this card could be any of the cards PLAYER 1 could theoretically have had in their hand at the time, and so the model therefore deduces (correctly) that PLAYER 2 could now have any of the cards that PLAYER 1 could have had. However, if the model knew that PLAYER 1 had had all four Aces, then after these moves, the model now thinks that it is possible for either the original player or the new player to have all four Aces, despite the fact that the new player can clearly have at most one Ace, as they only picked up one card. As a result, if PLAYER 2 were to play two Aces now, the model would wrongly accept this as possible.

This problem comes as a result of tracking individual cards rather than directly tracking every possible combination of cards that each player could have, as it is indeed possible for the new player to have any one of the four Aces. However, we unfortunately cannot directly track all possible combinations of cards, as the number of possible combinations is massive. It would be entirely infeasible both to keep track of this and to query it to see whether any possible combination would allow the move made. However, while this model does not have 100% recall when detecting moves which are literally impossible, it does have 100% precision.

Interestingly, taking advantage of this model resulted in barely any improvement compared to TIMID, with no statistically significant difference between the two.

4.4 NAIVE++

My next plan was to add in a couple of small changes of the sort that most human players adopt fairly quickly. The first was simply to sometimes change how many

1 v 3

	RANDOM5	TIMID	NAIVE
OPTION 2	29287	20205	17717
OPTION 3	29360	20265	17703

3 v 1

	RANDOM5	TIMID	NAIVE
OPTION 2	39731	36063	35018
OPTION 3	39617	35721	34615

Table 4.1: Number of games won by the OPTION 2 or 3 agent out of 40000 against each other agent. The null hypothesis predicts 10000 wins.

cards to play when cheating (though still only cheating if absolutely necessary). I chose a probability distribution for how many cards to play of $P(1) = 0.5$, $P(2) = 0.4$, $P(3) = 0.1$. Notably, the agent never cheats with 4 cards, as at least one other player must have in their hand or have played that round at least one card of the value claimed, so would always be able to deduce that they should challenge.

The second was a heuristic to choose which value to claim while cheating. This calculates an 'uncertainty' score for each card value based on the current player's belief model, then chooses the value to claim based on this. Uncertainty is calculated by summing the number of cards of that value that this agent believes each other agent could possibly have in their hand. For example, if this agent knows it has one Ace, PLAYER 2 has two Aces, and the other Ace could be anywhere, the uncertainty would be $3+1+1=5$, whereas if it knows nothing, the uncertainty would be $4+4+4=12$.

Initially, the agent claimed the value with the maximum uncertainty, however this in fact lead to a decrease in performance against both NAIVE and TIMID. This is because the uncertainty score is based on the current player's beliefs of where the cards are, not the other players'. In particular, this means that the current player is less likely to choose a value it has already played this round, despite the fact this is a common strong move played by high level human players (bluffing whether they cheated this time or last time).

I identified three possible ways of attempting to fix this problem: attempting to model all other players' expected uncertainty scores; first prioritising values already played, or using the uncertainty scores if that was not an option; or simply picking the minimum value. I ruled out the first option as I felt the challenge involved in such detailed modelling would not be worth the likely relatively small increase that it could provide over the other systems. I hypothesised that the second option would then be better than the third, both in terms of results against other agents and in acting more like a human would, but I implemented both to see which performed better against other agents. In these experiments, they actually both performed at a fairly similar level (see Table 4.1 for full results), so I chose the second option for more closely resembling human play.

Overall, these two simple heuristics lead to a dramatic improvement, with NAIVE++ beating each of the three previous agents with a win rate of over 42%.

4.5 CUNNING

The next step was to start cheating even when not required. In the CUNNING agent I kept this simple: on its turn, even if it is able to play honestly, it will cheat with probability p, a parameter which is set on initialisation. I will use CUNNING_p to mean a CUNNING agent with probability of cheating when not forced to of p.

I also added two further rules to help improve the agent's performance. Firstly, if it is possible for the agent to win this turn without cheating, then it will always do so rather than randomly choose whether to cheat or not. Secondly, it will choose which cards to play from its hand when cheating based on a fairly simple heuristic, again one often used by human players: prioritise playing cards with values for which it has the least cards of that value in its hand. In other words, if it has only one Ace and one King, it will prioritise cheating with those two cards rather than two Queens. This is a reasonably sensible strategy, as when actually playing honestly, it allows the agent to on average play more cards each turn. If there is a tie between these cards, it will prioritise those whose values are furthest away from the current value. Again, this is a reasonable strategy, as unless someone challenges, the legal values on its next turn will be close to the current value, so this increases your odds of being able to play honestly (or cheat convincingly).

I experimented with various different values of p to see which values performed best against NAIVE++ agents. A graph of the results can be seen in 4.1, while full results can be found in Appendix A.1. Interestingly, probabilities close to one proved the most effective, showing that against a cautious challenger, the more you cheat, the better. This is not surprising, as when this tactic is successful it allows the agent to get rid of their cards much faster. Also of note is that even with $p = 0$, CUNNING_0 performs better than NAIVE++ due to its advanced heuristics for choosing cards to cheat with.

4.6 SUSPICIOUS

In order to further improve on these agents, I decided to improve the world model it stores. Rather than storing all the theoretically possible positions of each card, it instead creates a belief model, maintaining a probability distribution f_c over each card c's location, where $f_c = [P(c \text{ is in this agent's hand}), P(c \text{ is in PLAYER 2's hand}), P(c \text{ is in PLAYER 3's hand}), P(c \text{ is in PLAYER 4's hand}), P(c \text{ is in discard pile})]$. Note that this captures all the information that was present in the previous world model, as all non-zero probabilities are equivalent to the NAIVE model's belief that that card could be in this location. Initially, the probability distribution for each card c is set to either $\pi_c = [1,0,0,0,0]$ if c has been dealt to this agent, or $\pi_c = [0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0]$ if not.

In order to update the belief state, SUSPICIOUS has an opponent model for how it assumes a typical player behaves. Note that it uses a static opponent model as opposed to a dynamic opponent model: it does not update this model based on the other players' observed actions. This opponent model consists of two assumptions:

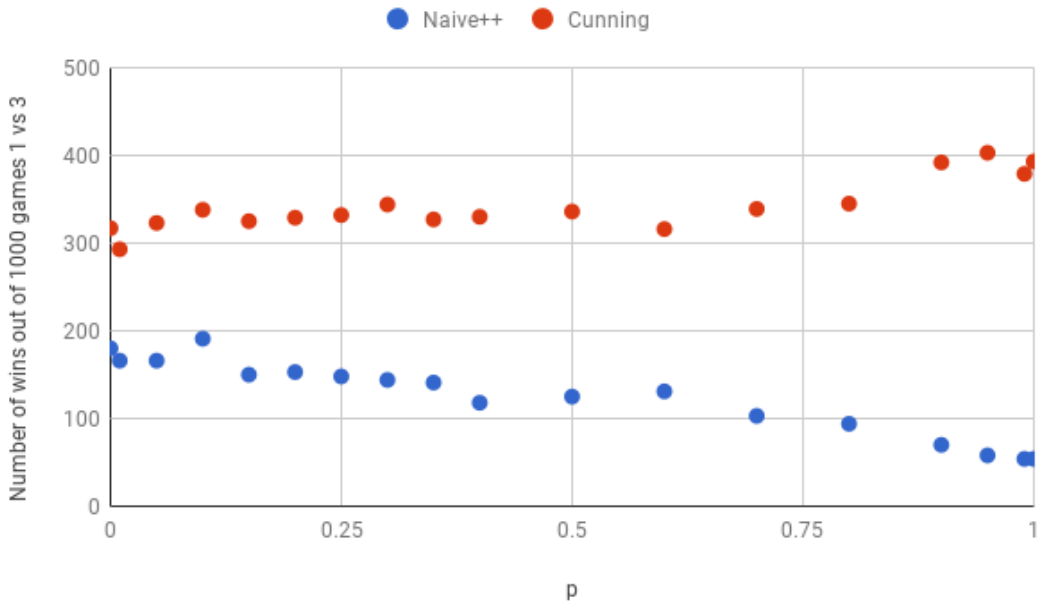

Figure 4.1: Performance of CUNNING$_p$ against NAIVE++ over 1000 games

1. Each opponent chooses to cheat with probability 0.1.
2. If cheating, the opponent picks which cards to play with equal probability.

Every move, SUSPICIOUS then updates the model according to this opponent model and its observations. Effectively, this means that whenever another player plays a card, the probability that any card is still in the other player's hand goes down by a small amount, while the probability that any card of the value claimed is now in the other player's hand goes down by a large amount, reflecting the fact that players are more likely to play honestly than to cheat. Similarly, for cards of the value claimed, the magnitude of this change depends on the number of cards of that value that could be in that players' hand, or for other cards, the magnitude depends on the total number of cards in that players hand. The probability that any of these cards are now in the discard pile then goes up by an equivalent amount.

More formally, upon the observation that player p has played n cards from a hand of size h with claimed value v, $\forall c$ with value $\neq v$:

$$\pi'_c[p] = \pi_c[p]\left(\prod_{i=0}^{n-1}\left(1-\frac{k}{h-i}\right)\right) \tag{4.1}$$

$$\pi'_c[4] = \pi_c[4] + \left(\pi_c[p] - \pi'_c[p]\right) \tag{4.2}$$

$$\pi'_c[i] = \pi_c[i], \forall i \in [0,3], i \neq p, \tag{4.3}$$

And $\forall c$ with value $= v$, taking r to be the number of cards x with value v for which $\pi_x[p] > 0$:

$$\pi'_c[p] = \pi_c[p]\left(\prod_{i=0}^{n-1}\left(1-\frac{1-k}{r-i}\right)\right) \tag{4.4}$$
$$\pi'_c[4] = \pi_c[4] + \left(\pi_c[p] - \pi'_c[p]\right) \tag{4.5}$$
$$\pi'_c[i] = \pi_c[i], \forall i \in [0,3], i \neq p, \tag{4.6}$$

Where π'_c is the updated probability distribution, and $k \in [0,1]$ is the fixed 'cheat probability' parameter from the player model.

Also, when a player p picks up all the discards, for each card, the probability it was in the discard pile gets moved to the probability of it now being in player p's hand. More formally, $\forall c$:

$$\pi'_c[p] = \pi_c[p] + \pi_c[4] \tag{4.7}$$
$$\pi'_c[4] = 0 \tag{4.8}$$
$$\pi'_c[i] = \pi_c[i], \forall i \in [0,3], i \neq p, \tag{4.9}$$

On the other hand, if this agent picks up all the discards, the probability distribution is reset to its value at the last pickup, then updated based on the exact value of the cards in the discard pile. In other words, the agent throws away its guesses for what everyone played, because now it knows for certain what they played.

Now that SUSPICIOUS has this improved world model, it can query it to ask what the probability of another agent having been able to make a particular move is, in other words what the probability is of them having x cards of the value v claimed. We use the formula described in [23] for the probability of at least m out of n independent events $(A_1 ... A_n)$ occurring:

$$P_m = S_m - \binom{m}{1}S_{m+1} + \binom{m+1}{2}S_{m+2} - \ldots \pm \binom{n-1}{m-1}S_n \tag{4.10}$$

where

$$S_k = \sum_{1 \leq i_1 < i_2 \ldots < i_k \leq n} P(A_{i_1} \times A_{i_2} \times \ldots \times A_{i_k}) \tag{4.11}$$

In our case, we have four independent events, namely the event that the card is in this players' hand for all four cards of this value, so $A_i = \pi_{c_i}[p]$ for each card c_i of value v. We can also take advantage of the fact that $P(\text{at least } x \text{ cards}) = 1 - P(\text{less than } x \text{ cards})$ to simplify some formulae.

So now SUSPICIOUS is able to accurately calculate the probability p that a move is possible based on its beliefs about the world. However, simply setting the policy to a mixed strategy of challenging with probability p and not challenging with probability $1-p$ is not a good tactic for a couple of reasons. First of all, if three similar SUSPICIOUS agents are considering challenging in this way, that means that the probability

of the challenge getting called will not be p but $1-(1-p)^3$, as there are three chances for the player to be challenged. This is clearly too high. Secondly, the fact that the other player has decided to play a particular value in fact gives us extra information which we are not using: players are more likely to play honestly, so by simply claiming to have these cards, the probability of this being true increases. This is not captured in the current model. Combining these two points makes it clear that we need to reduce the probability of challenging in some way, so the policy SUSPICIOUS follows is to challenge if the probability is below a certain threshold. This threshold is a parameter which can be set at initialisation time, so this solution also neatly introduces a 'trust' parameter to the SUSPICIOUS agent. Similar to previous examples, we write SUSPICIOUS$_{x,y}$ to mean a SUSPICIOUS agent with probability of choosing to cheat of x and probability threshold for challenging of y.

After running many simulations over different values, SUSPICIOUS$_{0.15,0.05}$ proved to be the most successful version. This managed to beat all other agents apart from NAIVE, despite the fact that all agents other than CUNNING cheat as little as possible.

4.7 BALANCED

There were a number of additional heuristics which I thought could prove effective based on my experience with playing the game myself.

4.7.1 Conditioning Cheating Probability on Discard Pile Size

First, I wanted to investigate changing the probability of choosing to cheat based on the number of cards in the discard pile. Since the penalty of picking up all the discards is bigger if the size of the pile is bigger, I hypothesised that choosing to cheat with probability inversely proportional to the size of the discard pile would improve play. To test this, I compared three agents with different functions $P(C|N=n)$, the probability of choosing to cheat (C) given the discard pile has n cards ($N=n$).

1. CUNNING$_p$'s behaviour was discussed above; its function is given by:

$$P(C|N=n) = p \tag{4.12}$$

2. EXPONENTIAL$_{p,k}$ is half as likely to cheat for every k cards in the discard pile. Formally, the function is:

$$P(C|N=n) = \frac{p}{2^{\frac{n}{k}}} \tag{4.13}$$

3. LINEAR$_p$'s cheating probability decreases linearly rather than exponentially, with the function:

$$P(C|N=n) = p\left(\frac{52-n}{52}\right) \tag{4.14}$$

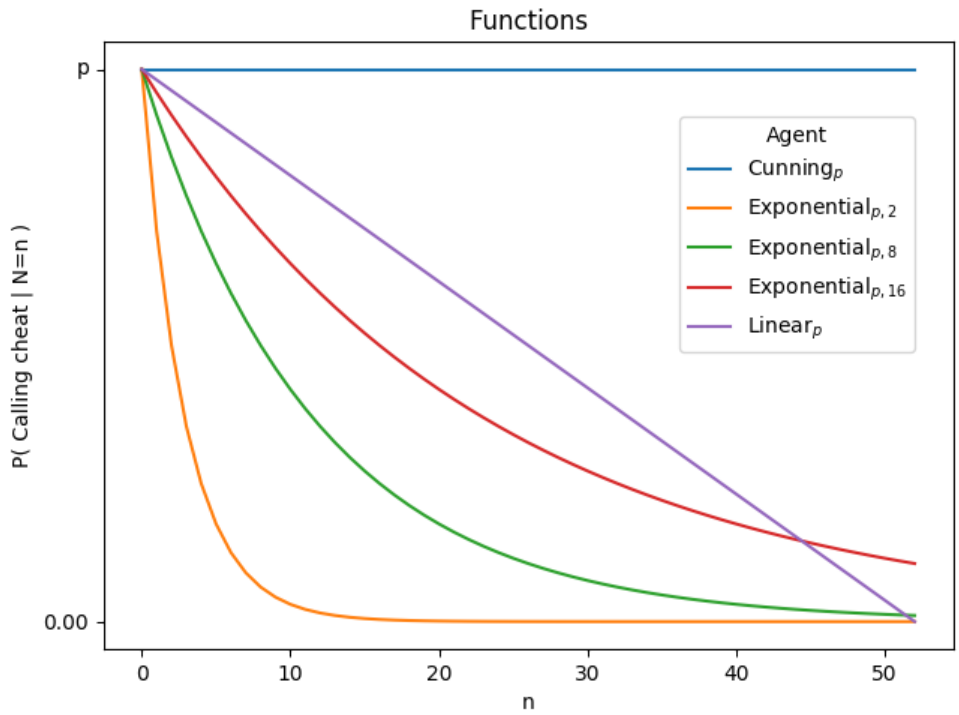

Figure 4.2: Comparison of Different Functions for Conditional Cheat Probability Based on Number Of Cards in the Discard Pile

A graph of each of these functions can be seen in Figure 4.2.

I tested the performance of these functions across various different values of p and k against SUSPICIOUS$_{0.15,0.05}$; full results are in Appendix A.2. The most successful function was EXPONENTIAL$_{0.3,16}$, however the difference is not large enough to carry much significance (p-value > 0.1). Nonetheless, I decided to use the EXPONENTIAL$_{0.3,16}$ function in BALANCED.

4.7.2 Uncertainty Calculation

The next heuristic I investigated was the uncertainty heuristic described in Section 4.4. Recall that the purpose of this heuristic is to estimate how uncertain other players are about the location of the cards of each value, then to pick the card value with the highest uncertainty when cheating with the intention of avoiding picking values which the other players know are not in your hand.

The function currently that proved most effective for NAIVE++ was to sum the number of possible cards each other player could have, then choose the minimum value. However, this does not take advantage of the updated belief model. I therefore designed a new uncertainty heuristic SUSPICIOUSUNCERTAINTY. For all cards of the inputted value, it converts the probability of an agent having had that card at the start of this round to an uncertainty score, then it sums the uncertainty over all these cards over all

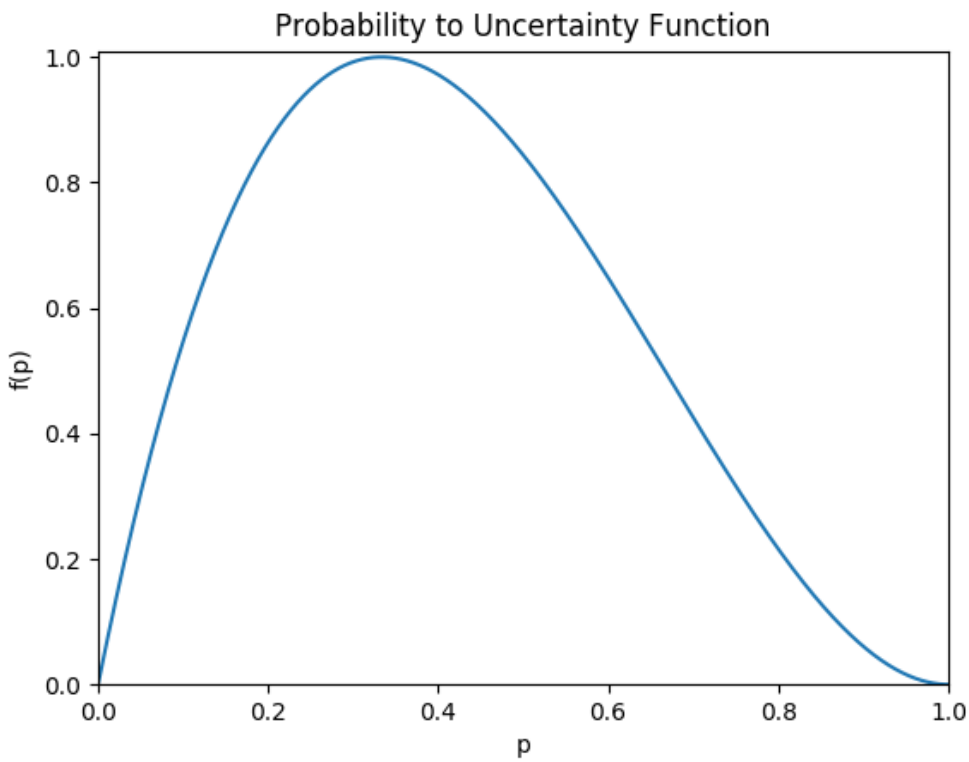

Figure 4.3: Graph of Probability to Uncertainty Conversion Function

other agents, and adds a constant if it had this card at the start of the round.

In order to convert from a probability to an uncertainty for one particular card, we want a probability of either 0 or 1 to return an uncertainty of 0, while a probability of $\frac{1}{3}$ should return a maximum uncertainty of 1. This conversion function must also be continuous and real valued over the range [0,1]. In other words, we require a function f for which:

- $f(0) = 0$
- $f(1) = 0$
- $f(\frac{1}{3}) = 1$
- $f'(\frac{1}{3}) = 0$
- $f''(\frac{1}{3}) < 0$
- $\forall p \in (0,1), p \neq \frac{1}{3} \iff f'(p) \neq 0.$

I chose the function $f(p) = \frac{9}{4}\left(p^3 - 2p^2 + p\right)$ to meet these parameters, as shown in Figure 4.3.

To evaluate this heuristic to see if it performs better than the old one, I tested SUSPICIOUS (old heuristic) against SUSPCIOUSUNCERTAINTY (new heuristic) in 1000

games. SUSPICIOUS won 201 games 1 v 3, while SUSPICIOUSUNCERTAINTY won 294 games 1 v 3, a clear improvement (p-value < 0.01).

4.7.3 Cheat Prediction

The final function I improved was the function which gives the probability of challenging a move (X) given the confidence that that player has those cards in their hand ($C = c$). I came up with three different options:

1. SUSPICIOUS is the baseline which we have been using so far, it challenges if its confidence that the player has those cards in their hand is less than its trusting threshold t. Note that this can be exploited fairly easily, for example it will never challenge a move which it knows you have the cards for, despite the fact you could nonetheless be cheating. Its function is given by:

$$P(C|X = x) = \begin{cases} 0 & x \geq t \\ 1 & x < t \end{cases} \tag{4.15}$$

2. EXPONENTIAL always challenges if the confidence is zero (i.e. $P(X|C = 0) = 1$, then as its confidence decreases, the probability of challenging will also decrease, based on two parameters. First, the base distrust k, the probability of challenging even when certain that that move was possible, $P(X|C = 1) = k$. Second, the midpoint j, the confidence level at which $P(X|C = j) = \frac{1}{2}$. The function which captures this is:

$$P(C|X = x) = \frac{1}{\left(\frac{1}{k} - 1\right) x^{\log_j\left(\frac{k}{1+k}\right)} + 1} \tag{4.16}$$

3. SIGMOID challenges based on a sigmoid function with minimum value k, maximum value m, midpoint j, and steepness s. In particular, $P(X|c = \infty) = k$, $P(X|c = -\infty) = m$, $P(X|c = j) = \frac{1-k}{2}$. Because a sigmoid's maximum and minimum values are asymptotic, it means that $P(X|c = 0) \neq 1$, though it clearly should be, and similarly $P(X|c = 1) \neq k$, so the function also includes special cases for these values. In other words, the function is:

$$P(C|X = x) = \begin{cases} 1 & x = 0 \\ k & x = 1 \\ \frac{m-k}{1+e^{s(x-j)}} + k & \text{otherwise} \end{cases} \tag{4.17}$$

A graph of each of these functions can be seen in Figure 4.4.

Surprisingly, testing these different functions showed that neither EXPONENTIAL nor SIGMOID performed better than the baseline SUSPICIOUS against $\text{SUSPICIOUS}_{0.15,0.05}$ for any parameters. However, SIGMOID almost managed to equal the baseline for some sets of parameters, for example $\text{SIGMOID}_{0,0.01,1,25}$. Full results can be found in Appendix A.3.

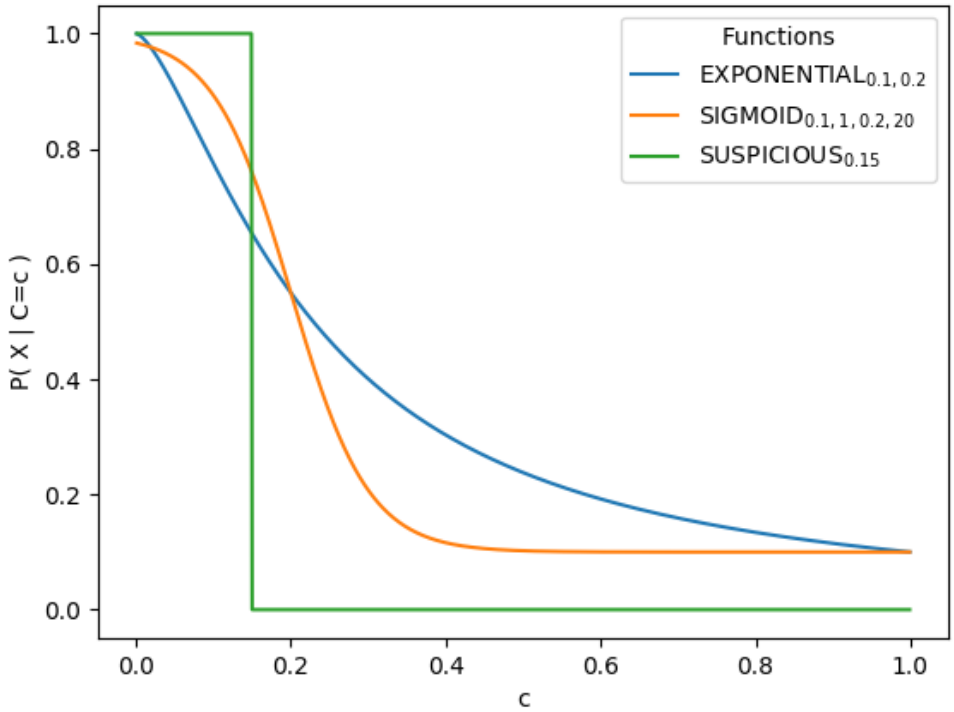

Figure 4.4: Graph of probability of challenging given confidence that that player has those cards

Ultimately, I decided to use the $\text{SIGMOID}_{0,0.01,1,25}$ function in the final BALANCED agent, as although it performed slightly worse against $\text{SUSPICIOUS}_{0.15,0.05}$ in the tests, it has the advantage of both being a mixed policy and being a more human-like policy than SUSPICIOUS, which were two of the aims described at the start of the section, so I decided this was worth the potential drop in performance.

4.8 Evaluation

The final results table showing a comparison amongst my rule based agents is shown in Figure 4.5. In general, each agent beats or equals every previous agent 1 v 3, with significance at at least the 0.990 level. The two exceptions to this rule are NAIVE and NAIVE++ which do not seem to fare much better than the basic TIMID agent. This suggests that due to the rapid increase in uncertainty in Cheat, trying to keep track of where each of the cards could possibly be does not offer that much help unless combined with an opponent model.

Another interesting takeaway is that when playing 1 v 3, SUSPICIOUS is able to comfortably beat any of the three agents which never choose to cheat, despite the fact that they act under a very different policy to SUSPICIOUS's opponent models for them. However, when playing 3 v 1, this incorrect opponent model appears to have a much larger negative impact on SUSPICIOUS's play, for TIMID in particular, resulting in a safe win for TIMID. This also provides a perfect example of the rock-paper-scissors behaviour I predicted in Section 3.3.4: TIMID beats SUSPICIOUS, SUSPICIOUS beats CUNNING, but CUNNING beats TIMID.

The table also makes it very clear which are the best and worst agents overall. RANDOM provides a very low baseline for any other agent to beat, while BALANCED fares reasonably well against all types of players. Testing BALANCED against human players also seems to suggest that BALANCED is a powerful opponent, though the sample size is too small to draw a strong conclusion from.

		Agent for players 2-4						
	Player 1 wins	Random_0.05	Timid	Naive	Naive++	Cunning_0.15	Suspicious_0.15,0.1	Balanced
Agent for player 1	Random_0.05	-	23	11	12	16	19	10
	Timid	515	-	235	223	130	293	185
	Naive	534	241	-	212	114	215	178
	Naive++	482	251	295	-	153	255	206
	Cunning_0.15	594	315	364	344	-	305	241
	Suspicious_0.15,0.1	717	287	373	405	292	-	232
	Balanced	759	331	382	410	292	297	-

Figure 4.5: The final results table. Each cell contains the number of games won by ROWPLAYER out of 1000 between one ROWPLAYER and three COLUMNPLAYERS. The null hypothesis predicts 250 wins.

Chapter 5

MCTS Approach

All the agents described so far have used rules and heuristics based purely on the current state to guide their play, without ever searching ahead in the game tree to improve their strategy. In this chapter, I describe the challenges involved with searching the game tree, the ways MCTS can avoid some of these challenges, and the ways of adapting MCTS to work in a partially observable environment like Cheat, as well as describing my efforts to build an agent which uses MCTS and reporting on its results.

5.1 Full Tree Search

As in any well defined game, it is theoretically possible to entirely solve Cheat (i.e. to produce an optimal policy) by simply performing a full search on the game tree from the current state to find the optimal action to take. However, the state space of the game is far, far too large to do this in practice.

Take a simplification of Cheat where the game state is always fully observable. The initial deal has $\binom{52}{13}\binom{39}{13}\binom{26}{13}\binom{13}{13} \approx 5.36 \times 10^{28}$ possibilities. Typical four-player games last around 50 game turns, with each turn consisting of between two and four plies (decision points), namely one player playing cards and claiming a value, then up to three others deciding whether to challenge. When playing cards from a hand of size n, you have $13(\binom{n}{4} + \binom{n}{3} + \binom{n}{2} + \binom{n}{1})$ possible options: for an average hand of size 13, this gives 14196 moves available. When deciding whether to challenge, you have 2 possible moves available. Putting this together, we get a very rough estimate of the size of an average game tree as $10^{28} \times 14196^{50} \times 2^{100} \approx 10^{266}$.

Clearly a tree of this size is far too large for any system to ever be able to naively search. However, notice that many of the states in that game tree would be duplicates. For example, playing the Ace of Clubs this turn and the Ace of Spades next turn would lead to the same state as if you played the Ace of Spaces this turn and the Ace of Clubs next turn but kept every other aspect the same. To get a more accurate idea of how large Cheat can actually be when we take duplicates into account, we can calculate the size of the Markov Decision Process (MDP), i.e. the number of possible states of the

game. Other than the location of each card, each state can have four different players whose turn it is to play cards or four different players whose turn it is to challenge three different other players, plus 14 different last values claimed (13 card values + no last turn), and two possibilities for whether the last move was or was not a cheat. As such, the number of states in the MDP $=$ (number of possible card arrangements)$(4+(4\cdot 3))\cdot 14\cdot 2$.

To calculate the number of possible card arrangements, observe that the discard pile can have zero or more cards, while every player's hand must have one or more cards, as otherwise the game is over. Also, the total number of cards between all locations must clearly add up to 52. This gives us a formula for the total number of possible card arrangements of:

$$\sum_{i=0}^{48}\left(\sum_{j=1}^{49-i}\left(\sum_{k=1}^{50-i-j}\left(\sum_{l=1}^{51-i-j-k}\binom{52}{i}\binom{52-i}{j}\binom{52-i-j}{k}\binom{52-i-j-k}{l}\right)\right)\right) \tag{5.1}$$

where i is the number of cards in the discard pile, and j, k, and l are the number of cards in player 0, 1, and 2's hands respectively. This is equal to roughly 4.38×10^{34}, giving us an overall state space size of 1.96×10^{37}. This is much smaller than the size of the tree when not dealing with duplication, but is still too large to hope to solve perfectly. Also, these calculations have all been for the fully observable Cheat; normal Cheat will have an even higher bound.

5.2 Monte Carlo Tree Search

This is where Monte Carlo Tree Search comes in. Monte Carlo Tree Search (MCTS) is a heuristic-based tree search algorithm which has recently seen outstanding success in a number of fields with infeasibly large search spaces, notably including AlphaZero's results in Go [1] and chess [24]. It is an online learning algorithm: rather than calculating the best action to take from every possible state in advance, it calculates only the best action to take from the states it actually experiences in the game. This has the advantage of allowing it to avoid over-generalising in environments with large state spaces, but has the disadvantage of not being able to take advantage of the fact that similar states often have similar strategies. It also needs to do all the computation at runtime rather than at training time, meaning the power of the algorithm is bounded by the thinking time available.

In its standard form, the algorithm consists of four repeated steps:

1. Selection: Choosing which state in the tree to explore further.
2. Expansion: Adding the child states of the selected state to the search tree and choosing one of these to explore further.

3. Rollout: Simulating a full game from this state according to some fixed policy (typically random).
4. Backpropogation: Using the result of the simulated game to update all ancestor states, increasing their expected value if the game is a win or decreasing it if not.

The best action to take can then be extracted by examining all the states at depth one and choosing the action which leads to the state with highest expected value.

In the selection step, the algorithm evaluates which state is best to explore based on a balance of high expected value and low visit count. This allows the search to exploit the areas of the tree which it believes are likely to be particularly fruitful, while still exploring the areas it is currently unsure about. I use the UCT (Upper Confidence Bound 1 applied to trees) algorithm throughout this paper as my selection policy, as it is efficient to calculate, provides a good balance between exploration and exploitation, and is typically highly effective in practice. [25].

5.3 Nondeterministic MCTS

Standard MCTS as described above requires the environment to be deterministic and fully observable, which is not the case for Cheat. There are three main ways of extending MCTS to support partially observable environments: Perfect Information MCTS, Information Set MCTS, and Belief Distributions.

1. Perfect Information MCTS (PIMCTS) [17], also known as Determinized[1] MCTS, samples n possible instances of the game state based on the observable state, runs the standard MCTS algorithm on each of these, then combines the results from each of these sampled instances, returning the action which performed best on average.
2. Information Set MCTS (ISMCTS) [26] searches a tree of information sets rather than game states.
3. Belief MCTS (BMCTS) [27] searches a tree of belief sets rather than game states.

I chose to investigate PIMCTS over the other two methods as it is the most prevalent method in the literature, with good results across many domains, notably including many other card games such as Poker [16], Dou Di Zhu [17], and Klondike [28].

5.4 PIMCTS

In order to implement an agent which uses PIMCTS to calculate the best action to take, three important things were required: the standard MCTS algorithm, the MDP

[1] For consistency with the existing literature, I use the Americanised spellings of “Determinized” and “Determinization” throughout this paper.

defining the game, and the sampling algorithm. I used an implementation of standard MCTS written by Mihai Dobre, described in [27], and implemented the MDP for Cheat myself. Despite the fact that the main game engine and the MDP do essentially the same job, I implemented the MDP separately, as this allowed me to optimise the MDP much more, which is vital given the amount of processing that is required for MCTS. In order to then integrate these together, I had to first understand how Dobre's code worked, then write my Cheat MDP in such a way that his MCTS algorithm can use it.

I then used this to create a PIMCTS agent within my Cheat implementation which samples n game states from its belief model, converts these to MDP states, passes each to the MCTS algorithm, reads the statistics which that predicts for each possible action, combines the statistics from each sample, then chooses the action with the overall highest expected win rate.

5.4.1 Using Opponent Modelling in the Sampling Algorithms

The initial sampling algorithm I designed simply loops through all the cards in each opponent's hand and the discard pile and randomly assigns each of them a suit and value pair which is currently unassigned. However, this fails to take into account any of the history of the game, such as the knowledge that the cards this agent played this round must be in the discard pile, or that a particular player definitely had a particular card at the start of the round.

To improve this, the agent can maintain a belief model of the world based on a static opponent model, in the same way as SUSPICIOUS and BALANCED, then use this to guide the sampling so that the probability of generating each sampled state is no longer uniform, but instead is proportional to each state's likelihood. This leads to more iterations executed with the more likely states, improving the expected value computed by MCTS [16].

One possible way of achieving this is to enumerate all possible states, calculate the likelihood of each according to the agent's belief model, then take a weighted sample [29] from these, using each state's likelihood as its weight. However, this is impossible in practice due to the number of possible states. If the four players have n_0 n_1, n_2, and n_3 cards respectively, the number of possible states is $\prod_{i=0}^{2} \binom{52 - \sum_{j=0}^{i}(n_i)}{n_{i+1}}$. At the start of the game for example, the number of possible states $\approx 8.45 \times 10^{16}$.

Instead, I decided to develop a stochastic algorithm for generating the state from the belief model. Here the challenge was to generate a state with probability roughly proportional to its likelihood in the belief model, while also ensuring the restrictions on the correct number of cards in each player's hand and the discard pile are held. If not for these restrictions, it would be possible to simply stochastically assign each card to a location based on its probability distribution in the belief model. The algorithm also needed to be highly efficient as it is called many times each turn; in particular its run time should not depend on the total number of possible states, since as we saw

above, there are far too many of them. The algorithm I designed is given in Algorithm 1.

Algorithm 1

```
function DETERMINIZESTATE
    Initialise each other player's hand and the discard pile to empty
    for each card not in our hand do
        Stochastically pick a location for this card based on its probability distribution in the model
        Add this card to that location
    end for
    while any location has the wrong number of cards do
        Pick any location with too many cards
        Stochastically pick a card from this location, picking cards less likely to be in this location with higher probability
        Pick a location to move the card to. Prioritise locations with too few cards over those with the correct number, and those with the correct number over those with too many. In the event of a tie, prioritise the location with the higher probability of this card being in that location.
        if chosen location has a non-zero probability of containing this card then
            Move this card from the first location to the second
        end if
    end while
    return card locations
end function
```

This algorithm now provides an efficient way of estimating a weighted sample from the belief model. Every sample it generates is consistent with the agent's world view, and states with a lower likelihood have a lower probability of being sampled.

I also needed a way of sampling the move an opponent has just made when it is the PIMCTS agent's turn to decide whether or not to challenge. One possible way of doing this is to randomly choose cards from the determinized opponent's hand; another would be to include the cards the opponent has just played as another possible location for cards in the DeterminizeState function. However, both of these fail to capture the assumption in our opponent model that moves are honest a much higher percentage of the time than they are dishonest, so the fact that the opponent has claimed a particular value actually increases the probability that the cards played are indeed of that value.

Instead, I developed a move sampling technique which first samples a game state using the algorithm above, then it chooses to believe that the player chose to cheat with probability *distrust*. If it then believes the player either chose to or had to play dishonestly, it samples n cards from their hand with uniform probability. If not, it samples n cards of the value claimed from their hand, again with uniform probability over all cards of that value in their hand. The full algorithm is given in Algorithm 2.

Algorithm 2

```
1: function DETERMINIZEMOVE(player, numCardsPlayed, claimedValue)
2:     sampledState ⇐ DeterminizeState()
3:     if number of cards in sampledState[player] of value claimedValue < numCardsPlayed or random() < distrust then
4:         for i ⇐ 0 to numCardsPlayed do
5:             card ⇐ pickCardAtRandom(sampledState[player])
6:             move ⇐ move + card
7:             sampledState[player] ⇐ sampledState[player] − card
8:         end for
9:     else
10:        for i ⇐ 0 to numCardsPlayed do
11:            card ⇐ pickCardAtRandom(getCardsWithValue(claimedValue, sampledState[player]))
12:            move ⇐ move + card
13:            sampledState[player] ⇐ sampledState[player] − card
14:        end for
15:    end if
16:    return move
17: end function
```

5.4.2 Reducing the Branching Factor

As discussed in Section 5.1, one of the challenges of doing any form of tree search for Cheat is the massive branching factor: $4(\binom{n}{4}+\binom{n}{3}+\binom{n}{2}+\binom{n}{1})$ when choosing to play cards from a hand of size n on a typical turn, or $13(\binom{n}{4}+\binom{n}{3}+\binom{n}{2}+\binom{n}{1})$ if playing on an empty discard pile (i.e. if able to claim any value). For even the starting hand of size 13, this gives more actions to explore than number of iterations carried out each turn, meaning we would not even explore every possible next action once, let alone take advantage of the power of MCTS to focus the search in areas of high value. To help combat this, I enforced a couple of extra restrictions on what actions PIMCTS could take:

1. When playing cards honestly, the agent must play all cards of that value from their hand.
2. If the agent chooses to cheat, they can cheat by playing one, two, or three cards, but not four cards.
3. After the agent has chosen to cheat and the number of cards to play, the cards they cheat with are selected automatically based on the same heuristic that CUNNING uses.

Adding these restrictions should not directly affect the agent's play significantly, as the actions still available are usually as good as or better than the similar banned actions. However, these restrictions massively reduce the branching factor down to a maximum of $3+3\cdot 3=12$ on a typical turn, or $13+13\cdot 3=52$ if playing on an empty discard pile, i.e. if able to claim any value. While still large, this is now actually feasible to

search through.

5.4.3 Improving the Rollout

One of the challenges I ran into when performing the determinized MCTS search is that using a uniform random policy for the rollout, as is common in the literature, does not work well for Cheat. This is because under a uniform random policy, the game length dramatically increases, to the extent that the games take too long to simulate. The problem lies in the fact that an agent is three times as likely to cheat as it is to play honestly, while also having a probability of being challenged of $1 - \left(\frac{1}{2}\right)^3 = \frac{7}{8}$. This results in lots of back and forth between the agents, with no player being able to get rid of their cards quickly, dragging out the game far longer than it ought to take.

To counteract this problem, I use a heavy playout [30] to bias the rollout policy to more closely approximate a real game; in particular, it increases the probability of playing honestly and decreases the probability of challenging. As well as fixing the problem of simulated game length, using a heavy playout has the added benefit of improving the expected value calculated by MCTS against a typical opponent [31].

5.5 Evaluation

Having implemented the PIMCTS agent, I then evaluated its performance in games against each of the stochastic rule-based agents described in Chapter 4. The full table of results is shown in Table 5.1

	PIMCTS 1 v 3	PIMCTS 3 v 1
$\text{RANDOM}_{0.05}$	98	95
TIMID	0	21
NAIVE	0	25
NAIVE++	0	20
$\text{CUNNING}_{0.15}$	0	41
$\text{SUSPICIOUS}_{0.15,0.1}$	2	11
BALANCED	5	30

Table 5.1: PIMCTS v each of the other agents over 100 games. Null hypothesis predicts 25 wins 1 v 3 and 75 wins 3 v 1.

As you can see, PIMCTS fares incredibly well against $\text{RANDOM}_{0.05}$, managing to win a higher proportion of games than any of the rule-based agents.. However, when playing against an opponent with an actual strategy, it is out of its league, and struggles to get a single win in in the 1 v 3 case. It does fight back marginally better in the 3 v 1 case, but it still fails to come anywhere close to the 75% win rate expected if the agents were equal.

There are a few reasons why PIMCTS does not perform particularly well in this environment.

5.5.1 Large Tree Size

As discussed in Section 5.1, the average game tree size for even a determinized game of Cheat is massive. Even after applying the simplifications and restrictions described in Section 5.4.2, the branching factor is still significant. The average restricted game tree size can then be estimated as $\left(\left(12\,(1-p)^3 + 52 \cdot \left(1-(1-p)^3\right)\right) \cdot (2 \cdot 3)\right)^n$ where p is the average probability of an agent challenging and n is the average game length. For $p = 0.1$ and $n = 50$, a fairly typical game, this gives an estimated average game tree of size 6.96×10^{106}. Although MCTS is designed to be a relatively effective way of searching trees of these sizes, the number of rollouts needed per determinized game in PIMCTS still has to be large in order to cope well with this. From the results, it appears that the number of rollouts needed to be effective is too large to be feasible within the time limits I set in Section 3.2.3.

5.5.2 Sampling

One of the major downsides of using PIMCTS is that the agent is effectively unable to learn its own strategy for whether to challenge a particular move. This is because when PIMCTS samples a state, it then gets to see the exact cards the other player just played, meaning the decision of whether to challenge becomes an obvious one. As a result, PIMCTS misses out on the opportunity to learn its own strategy, as its policy for challenging simply becomes 'Challenge with probability equal to the probability of sampling a state where the other agent was unable to play honestly'. However, not knowing the opponent's cards and still reacting to that uncertainty appropriately is a major part of the strategy for Cheat which a good agent should be able to learn.

5.5.3 Strategy Fusion and Non-locality

Two significant weaknesses of PIMCTS which arise in some domains are strategy fusion and non-locality.

Strategy fusion [33] occurs when PIMCTS incorrectly believes it can use a different strategy in the future of each of the sampled states, but once it actually gets to that state, it has not yet learned which it is in so cannot make the informed decision it had planned. An example is given in Figure 5.1. In this, the upward pointing triangles represent states in which it is Player 1's turn, the downward pointing triangles represent states in which it is Player 2's turn, and the squares represent terminal states with the value for Player 1 given below. Neither player knows which of the two possible worlds they are in at any point until reaching a terminal state. To perform the search, Player 1 samples one of the two possible worlds for this search instance. No matter which world was sampled, Player 1 believes they have a winning strategy should they take the left action from the root state: by later moving to a if they sampled World 1, or b if they sampled World 2. As such, they will deduce that moving left from the root node is a guaranteed win for them in each sampled case, and therefore on average

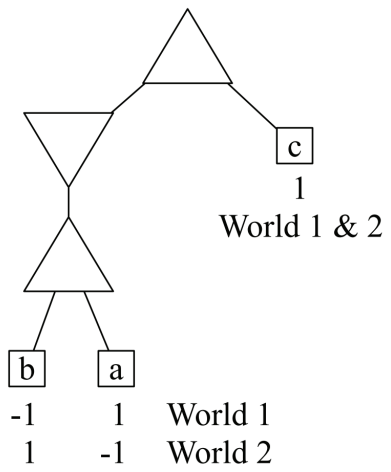

Figure 5.1: Example of strategy fusion, adapted from [32]

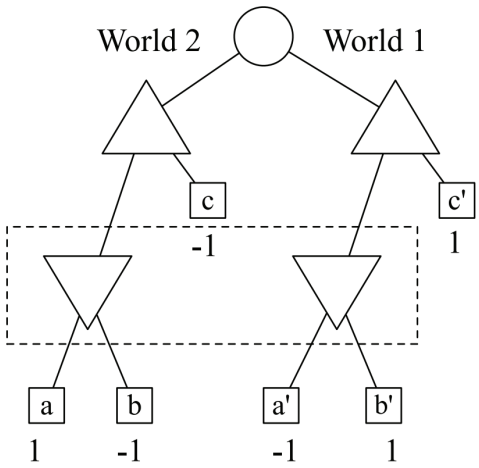

Figure 5.2: Example of non-locality, adapted from [32]

too. However, if they were to choose this action, they would find themselves unable to determine which world they were in for their next move, giving them an expected reward of 0, not the 1 that they predicted.

Non-locality [33] occurs when the value of a particular state depends on part of the game tree not contained within its subtree, usually due to deductions over actions taken by other agents based on private information. An example is given in Figure 5.2, where again the upward pointing triangles represent states in which it is Player 1's turn, the downward pointing triangles represent states in which it is Player 2's turn, the squares represent terminal states with the value for Player 1 given below, the circle represents a chance node, and the dotted rectangle represents an information set: a set of states which the acting player cannot differentiate between. When searching this tree, PIMCTS samples one of the two worlds at random to determine Player 2's action, resulting in it evaluating the left branch as having value 0. In reality though, if Player 2 gets to make a move, they can deduce they must be in World 2, as otherwise Player 1 would have simply picked c' and won, and therefore Player 2 will always pick b,

giving Player 1's left action an actual value of -1.

Long et al. [32] describe three measurable properties of an environment which can be used to estimate the likelihood of strategy fusion and non-locality causing problems for a PIMCTS search. These are:

1. Leaf correlation: The probability that all sibling terminal states have the same value.
2. Bias: The probability that the game favours one player over another; e.g. in a two player game, this is equal to the proportion of all terminal states for which Player 1 wins.
3. Disambiguation factor: The rate at which the number of states in each player's information set decreases relative to the depth of the tree.

I investigated the values of these properties for Cheat in order to see how much of a negative impact strategy fusion and non-locality are likely to have on PIMCTS's behaviour.

We can see that Cheat has leaf correlation of 1, as the only time two terminal states are siblings is when a player has played their last cards honestly and the final player is deciding whether to challenge that move: whether they challenge or not, the other player will win.

In order to calculate the bias, I simulated 80000 games between four identical agents, varying the type of agent playing each time. Out of these games, the agent who played first won 21269, second won 21400, third won 19373, and fourth won 17931. This gives an estimated bias of between 0.27 and 0.28.

As for the disambiguation factor, when playing Cheat the number of states in each player's information set actually increases from turn to turn, only decreasing when that player picks up the discard pile. This means that the disambiguation factor will be very low, estimated at between 0 and 0.01.

I compare these values to the experiments done by Long et al. in Figure 5.3. It is clear that this suggests that PIMCTS is greatly impacted by instances of strategy fusion and non-locality.

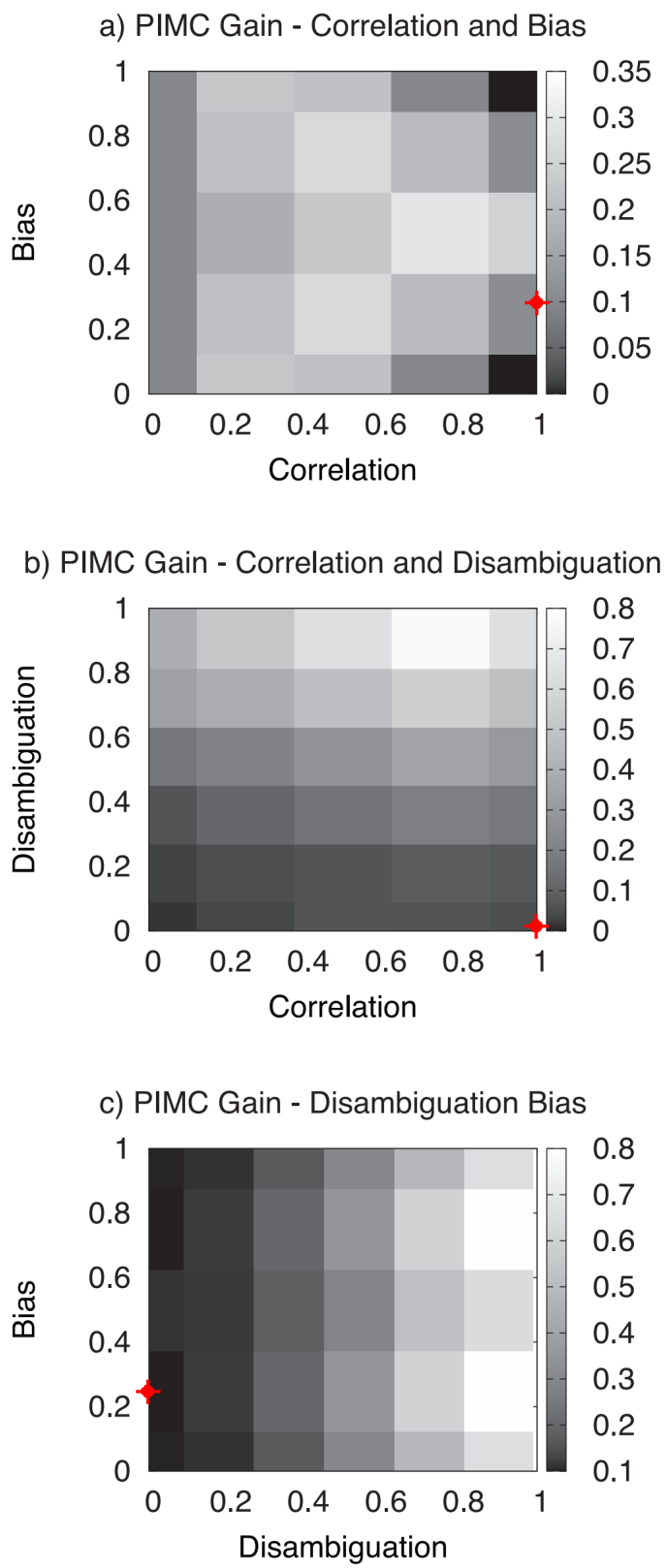

Figure 5.3: Performance gain of PIMCTS over a random agent against a Nash equilibrium agent for synthetic games with specific leaf correlation, bias and disambiguation factors, with the estimated values for Cheat plotted in red. Lighter regions correspond to increased performance of PIMCTS over random. Disambiguation is fixed at 0.3 in graph a, bias at 0.5 in graph b, and correlation at 0.75 in graph c. Figure adapted from [32]

Chapter 6

Conclusion

6.1 Summary and Evaluation

I have designed and implemented a game environment for Cheat, and have developed seven different stochastic rule-based agents and evaluated them against each other. The best of these, BALANCED, is a very strong player, able to beat even expert humans. However, none of these agents use dynamic opponent modelling to adapt their strategy based on their opponent, meaning if an opponent were to find a weakness in their strategy, they can exploit this indefinitely.

I have also investigated how effective PIMCTS is in tackling the complexities of Cheat. My experiments showed that unlike in other similar domains such as Skat [19] and Poker [15], PIMCTS struggles to perform at a high standard in Cheat. I identified a number of reasons why this is the case, most notably the large negative impact that strategy fusion and non-locality have when determinizing Cheat's game tree, largely due to Cheat's very low disambiguation factor.

6.2 Future Work

I think there are two significant areas where BALANCED could still be improved. The first I have discussed in some detail already, and that is for the agent to maintain a model for each of its opponents' strategies, rather than just assuming a default strategy for all other players. This allows it to improve the accuracy of its belief model, leading to a general increase in performance, as well as to change its strategy if the opponent finds and starts exploiting an existing weakness, keeping its performance more stable against all different style of play.

The second improvement that I believe could make BALANCED perform even better would be to change the three functions discussed in Chapter 4.7 (probability of cheating, calculating uncertainty of a value, and predicting whether the opponent is cheating) from hand-designed functions to entirely agent-learned functions using a su-

pervised learning technique such as a neural network [34]. This has the potential to train a function with much higher precision and recall than the limited number of functions I chose to investigate.

There is also still a lot of potential in exploring MCTS-based approaches to Cheat. For example, unlike PIMCTS, ISMCTS is not susceptible to strategy fusion or non-locality [26], so it still has great potential in this domain. Another potential system which could have more success than PIMCTS is Belief MCTS (BMCTS), as that keeps all the uncertainty about the state of the world directly in the tree, again preventing it from losing accuracy due to strategy fusion and non-locality, while also using information from the agent's belief model to guide the search.

Bibliography

[1] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[2] Tanguy Chouard. The Go Files: AI computer wraps up 4-1 victory against human champion. *Nature News*, 2016.

[3] BBC. Google achieves AI 'breakthrough' by beating Go champion. `http://www.bbc.co.uk/news/technology-35420579`, 2016. [Online; accessed 8-April-2018].

[4] Shih-Chieh Huang, Broderick Arneson, Ryan B. Hayward, Martin Müller, and Jakub Pawlewicz. MoHex 2.0: A Pattern-Based MCTS Hex Player. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games*, pages 60–71, Cham, 2014. Springer International Publishing.

[5] Richard J Lorentz. Experiments with Monte-Carlo Tree Search in the Game of Havannah. *ICGA Journal*, 34(3):140–149, 2011.

[6] Tristan Cazenave. Nested Monte-Carlo Search. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 456–461, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[7] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[8] Hubert Phillips. *The Pan Book of Card Games*. Pan Books, 1960.

[9] John McLeod. Rules of Card Games: Bullshit / Cheat / I Doubt It. `https://www.pagat.com/beating/cheat.html`, 2016. [Online; accessed 8-April-2018].

[10] Tina Philip. BLUFF WITH AI. Master's thesis, San José State University, 2017.

[11] Yotam Hacohen and Michael Doron. J'accuse: A game of Deceit and Accusations.

[12] Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[13] Ronen Tamari and Erez Peterfreund. Applying Reinforcement Learning to the card game of Cheat.

[14] Sander C.J. Bakkes, Pieter H.M. Spronck, and H. Jaap van den Herik. Opponent modelling for case-based adaptive game AI. *Entertainment Computing*, 1(1):27 – 37, 2009.

[15] Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-Carlo Tree Search in Poker using Expected Reward Distributions. In *Asian Conference on Machine Learning*, pages 367–381. Springer, 2009.

[16] Marc Ponsen, Geert Gerritsen, and Guillaume Chaslot. Integrating Opponent Models with Monte-Carlo Tree Search in Poker. In *AAAI Workshops*, 2010.

[17] Daniel Whitehouse, Edward Jack Powley, and Peter I Cowling. Determinization and Information Set Monte Carlo Tree Search for the Card Game Dou Di Zhu. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 87–94. IEEE, 2011.

[18] Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Malaysia; Pearson Education Limited,, 2016.

[19] Michael Buro, Jeffrey Richard Long, Timothy Furtak, and Nathan R Sturtevant. Improving State Evaluation, Inference, and Search in Trick-Based Card Games. In *ICJAI*, pages 1407–1413, 2009.

[20] Sam Ganzfried and Tuomas Sandholm. Game Theory-Based Opponent Modeling in Large Imperfect-Information Games. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 533–540. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[21] H Jaap van den Herik, HHLM Donkers, and Pieter HM Spronck. Opponent Modelling and Commercial Games. In *Proceedings of the IEEE*, pages 15–25. Citeseer, 2005.

[22] Karl Pearson. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[23] William Feller. *An Introduction to Probability Theory and its Applications*. John Wiley & Sons, 1968.

[24] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, abs/1712.01815, 2017.

[25] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Ma-*

chine Learning: ECML 2006, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[26] Peter I Cowling, Edward J Powley, and Daniel Whitehouse. Information Set Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):120–143, 2012.

[27] Mihai Dobre. *Low Resource Learning in Complex Games*. PhD thesis, The University of Edinburgh, 2018.

[28] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In *Proc. 19th Int. Conf. Automat. Plan. Sched.*, pages 26–33, 2009.

[29] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181 – 185, 2006.

[30] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[31] Peter D. Drake and Steve Uurtamo. Move Ordering vs Heavy Playouts: Where Should Heuristics be Applied in Monte Carlo Go. In *Proc. 3rd North Amer. Game-On Conf.*, pages 35–42, Gainesville, Florida, 2007.

[32] Jeffrey Richard Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 134–140, 2010.

[33] Ian Frank and David Basin. Search in games with incomplete information: a case study using Bridge card play. *Artificial Intelligence*, 100(1):87 – 123, 1998.

[34] Jrgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

Appendix A

A.1 Optimal CUNNING Probability against NAIVE++ Tests

p	NAIVE++ wins 1 v 3	CUNNING$_p$ wins 1 v 3
0	180	317
0.01	166	293
0.05	166	323
0.1	191	338
0.15	150	325
0.2	153	329
0.25	148	332
0.3	144	344
0.35	141	327
0.4	118	330
0.5	125	336
0.6	131	316
0.7	103	339
0.8	94	345
0.9	70	392
0.95	58	403
0.99	54	379
1	54	393

Table A.1: CUNNING$_p$ v NAIVE++ over 1000 games. Null hypothesis predicts 250 wins.

A.2 Conditioning Cheating Probability on Discard Pile Size Tests

p	k	SUSPICIOUS$_{0.15,0.05}$ wins 1 v 3	EXPONENTIAL$_{p,k}$ wins 1 v 3
0.01	2	266	227
0.01	8	293	189
0.01	16	275	232
0.05	2	277	223
0.05	8	278	219
0.05	16	269	219
0.1	2	268	239
0.1	8	260	232
0.1	16	256	235
0.2	2	275	223
0.2	8	266	235
0.2	16	253	259
0.3	2	248	255
0.3	8	228	256
0.3	16	226	262
0.4	2	232	263
0.4	8	233	257
0.4	16	244	259
0.5	2	251	252
0.5	8	263	240
0.5	16	281	259

Table A.2: SUSPICIOUS$_{0.15,0.05}$ v EXPONENTIAL$_{p,k}$ over 1000 games. Null hypothesis predicts 250 wins.

p	SUSPICIOUS$_{0.15,0.05}$ wins 1 v 3	LINEAR$_p$ wins 1 v 3
0.01	270	239
0.05	267	240
0.1	237	251
0.2	245	263
0.3	245	276
0.4	246	237
0.5	275	246

Table A.3: SUSPICIOUS$_{0.15,0.05}$ v LINEAR$_p$ over 1000 games. Null hypothesis predicts 250 wins.

p	SUSPICIOUS$_{0.15,0.05}$ wins 1 v 3	CUNNING$_p$ wins 1 v 3
0.01	279	248
0.05	287	229
0.1	264	257
0.2	223	251
0.3	229	255
0.4	283	260
0.5	306	228

Table A.4: SUSPICIOUS$_{0.15,0.05}$ v CUNNING$_p$ over 1000 games. Null hypothesis predicts 250 wins.

A.3 Cheat Prediction Function Tests

t	SUSPICIOUS$_{0.15,0.05}$ wins 1 v 3	SUSPICIOUS$_{0.15,t}$ wins 1 v 3
0.05	254	246
0.1	298	203
0.2	327	154
0.3	310	65
0.4	372	20
0.5	365	6

Table A.5: SUSPICIOUS$_{0.15,0.05}$ v SUSPICIOUS$_{0.15,t}$ over 1000 games. Null hypothesis predicts 250 wins.

k	j	SUSPICIOUS$_{0.15,0.05}$ wins 1 v 3	EXPONENTIAL$_{k,j}$ wins 1 v 3
0	0.05	563	166
0	0.1	543	165
0	0.15	542	172
0	0.2	579	170
0	0.3	580	154
0	0.4	570	154
0.01	0.05	284	175
0.01	0.1	355	156
0.01	0.15	379	118
0.01	0.2	373	83
0.01	0.3	369	48
0.01	0.4	382	21
0.05	0.05	392	136
0.05	0.1	406	97
0.05	0.15	382	70
0.05	0.2	397	60
0.05	0.3	417	25
0.05	0.4	396	13
0.1	0.05	418	93
0.1	0.1	422	63
0.1	0.15	447	54
0.1	0.2	418	38
0.1	0.3	395	27
0.1	0.4	398	14

Table A.6: SUSPICIOUS$_{0.15,0.05}$ v EXPONENTIAL$_{k.j}$ over 1000 games. Null hypothesis predicts 250 wins.

k	j	m	s	SUSPICIOUS$_{0.15,0.05}$ wins 1 v 3	SIGMOID$_{k,m,j,s}$ wins 1 v 3
0	0.01	0.8	25	285	201
0	0.01	0.8	10	332	179
0	0.01	0.9	10	324	190
0	0.01	0.9	25	301	228
0	0.01	1	10	329	175
0	0.01	1	25	260	234
0	0.05	0.8	10	330	192
0	0.05	0.8	25	291	191
0	0.05	0.9	10	333	164
0	0.05	0.9	25	278	233
0	0.05	1	10	308	186
0	0.05	1	25	307	212
0.01	0.01	0.8	10	333	216
0.01	0.01	0.8	25	286	223
0.01	0.01	0.9	10	324	196
0.01	0.01	0.9	25	290	190
0.01	0.01	1	10	307	173
0.01	0.01	1	25	278	212
0.01	0.05	0.8	10	341	176
0.01	0.05	0.8	25	286	230
0.01	0.05	0.9	10	315	168
0.01	0.05	0.9	25	299	199
0.01	0.05	1	10	323	152
0.01	0.05	1	25	306	202
0.05	0.01	0.8	10	334	138
0.05	0.01	0.8	25	309	176
0.05	0.01	0.9	10	371	156
0.05	0.01	0.9	25	313	181
0.05	0.01	1	10	356	150
0.05	0.01	1	25	321	178
0.05	0.05	0.8	10	365	159
0.05	0.05	0.8	25	353	153
0.05	0.05	0.9	10	371	128
0.05	0.05	0.9	25	334	155
0.05	0.05	1	10	362	143
0.05	0.05	1	25	300	159

Table A.7: SUSPICIOUS$_{0.15,0.05}$ v SIGMOID$_{k,m,j,s}$ over 1000 games. Null hypothesis predicts 250 wins.