Universiteit
Leiden
The Netherlands

Bachelor Computer Science

Monte Carlo methods and algorithms
for the card game Hanamikoji

Nour Hassan

Supervisor:
M.J.H. van den Bergh

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

01/07/2025

Abstract

HANAMIKOJI is a card game for 2 players where the player must attempt to gain more "Victory Marks" and "Charm Points" than their opponent. The game ends when either player has obtained a majority of either, or after 3 rounds, in which case the winner is decided by a tie-breaker. Between rounds, the deck is reshuffled and the cards are dealt again. This research aims to apply various algorithms based on the Monte Carlo system to discover which methodologies are ideal, how various other techniques may further improve player accuracy, emergent patterns in computer play, and the nature of the game's state space.

Contents

1 Introduction

The Monte Carlo system is a methodology for sampling outcomes in probabilistic situations. Rooted in gambling and initially utilized primarily in nuclear weapons research [KW09], it has become a popular method for discovering optimal plays in games with incomplete information.
To give an example of such, take the card game Poker [Skl99]. All a player knows is the cards in their current hand. They do not know the hands of the other players, or the cards currently still in the deck. Based on this information, they need to decide their next move. If they have a strong hand, they will often opt to raise the bets. If they anticipate that another has a hand that exceeds their own, they will be far more inclined to check or even fold.
All decisions are based on human intuition. An approach based on the Monte Carlo system would be to play a set of completely random games, where the only constant between all of them is the hand of one player. Based on the outcomes, the aforementioned player can forego intuition and choose the play with the highest sampled probability of success. In the same vein, Hanamikoji [Hmk17] is also a card game with incomplete information. The game was first released in 2013, making it relatively new. The game, played by two people, has many variable aspects concealed from either player, such as the hand of the opponent, the cards in the deck, etc. Many different methods of random sampling are applicable for this.

In this research, we analyze some of these methods, observe their performance in the game, and analyze trends within the results. We also analyze potential correlations between various circumstances and game outcomes for the players. Hanamikoji is quite unexplored in terms of optimal play, making the efficacy of Monte Carlo methodologies in the game a compelling topic to research. The game has various traits worth analyzing, such as the relative value of individual cards and the impact of turn order. Another interesting attribute is the nature of the game's state space, and how that may influence the utility of various algorithms.

Figure 1: The cards in the deck, sorted by type.

Thesis Goal

In this research, the primary goal will be to discover which methodologies and supplementary techniques can be applied to achieve the most consistently optimal play within Hanamikoji. This necessitates not only maximizing the size of a search but the information gain thereof as well, so as to improve the quality of play while maintaining efficient runtimes. These techniques include regular Monte Carlo methods, the use of pre-sampled information as well as complete information of the current game state, recursively sampling multiple turns of the game at once, and lastly, the MCTS and Minimax algorithms.
The secondary goal is to analyze trends within the game itself, such as the impact of turn order, the average length of games, and the relative values of cards as opposed to their real value as described by the game's rules. Through this, patterns in play could translate to actual strategies in games between human players.
The final goal is to examine the nature of the state space of Hanamikoji. If the state space is too small, broad, or shallow, MCTS methods may prove ineffective as opposed to simpler algorithms. The underlying objective here is to observe whether the Trade-Off between the more expansive searches of MCTS justifies the potential improvement in the quality of play.

Thesis Overview

Firstly, we will highlight related works in Section 2. In Section 3, we will subsequently outline the rules and modifications of the simulated games of Hanamikoji we will be researching. In Sections 4, 5 and 6, we will observe how the game is simulated and interacted with, how agents use the aforementioned interactions to evaluate the decisions they make, and what underlying algorithms and techniques are used specifically for this process. In Section 7, we will outline what parameters the agents we will be testing can be configured with. In Section 8, we will observe the results of the experiments performed in accordance with the goals stated prior. Lastly, in Section 9, we will draw our final conclusions and speculate on what further research the results may invite.

2 Related Work

The Monte Carlo system is a popular avenue for optimization problems in game theory. In this section, we will explore several examples of it being applied in research on famous games. We will primarily examine applications in games meant for two players.

Monte Carlo Tree Search (henceforth referred to as MCTS) [BPW+12] methodology has been used to train models to play competitive Pokémon battles [Nor19]. In this research, the oldest games are used, presumably for their mechanical simplicity and lack of variance. In battles, each player's team and move choices are hidden until used in battle. The most relevant discovery in the paper is an ostensible causal link between the depth of the agent's searches and the average length of matches it played, as well as how often it won.

MCTS has also been employed for Poker [VdBDR]. In this instance, it applies a variation of the Minimax algorithm [DM] to simulate games played between random and AI players, with each layer of nodes representing the possibilities of moves in a turn. This paper explores key facets of MCTS such as the Trade-Off between accuracy and sampling time and the balance between exploration and exploitation (therein implemented through an Upper Confidence Bound [CM07]).

This methodology has been researched in games with no variance or obscured information as well. One popular example is Chess [Are22]. In a similar fashion to the Poker research, a Minimax algorithm is applied to examine which board states are most compelling to the player and their opponent on each turn. Since the state space of Chess is far too large to fully examine, MCTS remains compelling to approximate the desirability of sub-trees that would be impossible to fully traverse. Another important concept of this paper is searching with heuristics. Because of the aforementioned state space, unconditional random sampling will either be tedious or inaccurate. In Chess, concepts like material advantage, king positions, and pawn structures can, to an extent, be evaluated programmatically to curtail this issue.

Another example of a game without variance or incomplete information is Go [GS11]. In this research, we see similar concepts from the prior examples in effect. Heuristic search is similarly applied to alleviate the hurdle of the enormous state space of Go. The Upper Confidence Bound algorithm is utilized to balance exploration and exploitation as well. Similar methods have been applied by Google in their AlphaGo AI model [Fu18]

An indirectly related example of the utility of MCTS is its application in the game Tetris [CZN11]. In the game, you can see the upcoming pieces as the next one falls onto the grid, which allows for move planning on the basis of a tree which can be deepened in tandem with the revealing of new pieces on the grid.

The Minimax algorithm can also be used without MCTS in a game where the state space is sufficiently small, such as Connect 4. In this case, random sampling is not as informative as simply deepening the search tree, and we see that Alpha-Beta pruning [KM75] can even further simplify said searches.

These examples highlight that MCTS methodology is not only suitable for Hanamikoji, but that it is versatile and can make use of various ancillary techniques to be optimized for specific use cases. Several of these concepts will be recurring within this research as well. They also allude to an overarching correlation between the nature of the state space and the efficacy of searching, sampling, and the balance between both.

3 Hanamikoji Rules

In Hanamikoji, six cards are dealt to two players each from a deck of 21, after which one card is disposed from the deck for the remainder of the round. Up to three rounds may be played. Before each player takes a turn, they draw another card from the deck. At the end of each round, each player should have eight cards for scoring and two more discarded. These cards will belong to one of seven suits or **Geishas**, each worth a certain number of **Charm Points** determined by how many cards that match them are in the deck in total (2, 2, 2, 3, 3, 4, 5). Whichever player scores the most cards of a certain type, earns the **Victory Marker** and Charm Points of the corresponding Geisha. The game ends if a player gets four Victory Markers or eleven Charm Points by the end of a round. If three rounds pass with no decisive winner, the player with the largest number of Victory Marks or Charm Points wins the round. In both cases, Victory Marks take priority in deciding the winner. In each round, four distinct actions must be taken across four turns (in no particular order):

- **Secret**: Conceal a card from your opponent for the remainder of the round. This card will be scored.
- **Trade-Off**: Conceal and discard two cards from your hand for the remainder of the round. They will not be scored.
- **Gift**: The player presents the opponent with three cards of their choice from their hand. The opponent chooses one to score for themselves. The other two are scored for the player who made the offer.
- **Compete**: The player presents the opponent with four cards of their choice from their hand, in two pairs. The opponent chooses one pair to score for themselves. The other pair is scored for the player who made the offer.

3.1 Modifications

In this research, some alterations were made to simplify programming and game logic for the agents.

- The "Compete" action does not work with two pairs of cards, but rather four separate cards of which any disparate pair may be chosen.
- When drawing cards from the opponent's offers, they are placed in the recipient's hand rather than their scored cards. Scored cards can not be used in future turns.
- Consequently, the players end every round with three cards still in their hand. These are simply scored alongside the other cards at the end of the round.
- The round starts by dealing seven cards to both players and dealing more at the end of each player's turn, rather than dealing six and dealing more at the start of their turn.

The potential impact of these modifications is unclear. Analysis thereof is outside the scope of this thesis. An example of a round of the simulated game can be found on the following page. Note that the suits of the card are numbered 0–6. Furthermore, the indices in the deck and each player's hand and scored cards represent the quantity of cards of a particular suit. The indices in Trade-Off, Secret and Card Offer represent one card each, the number corresponding to the suit.

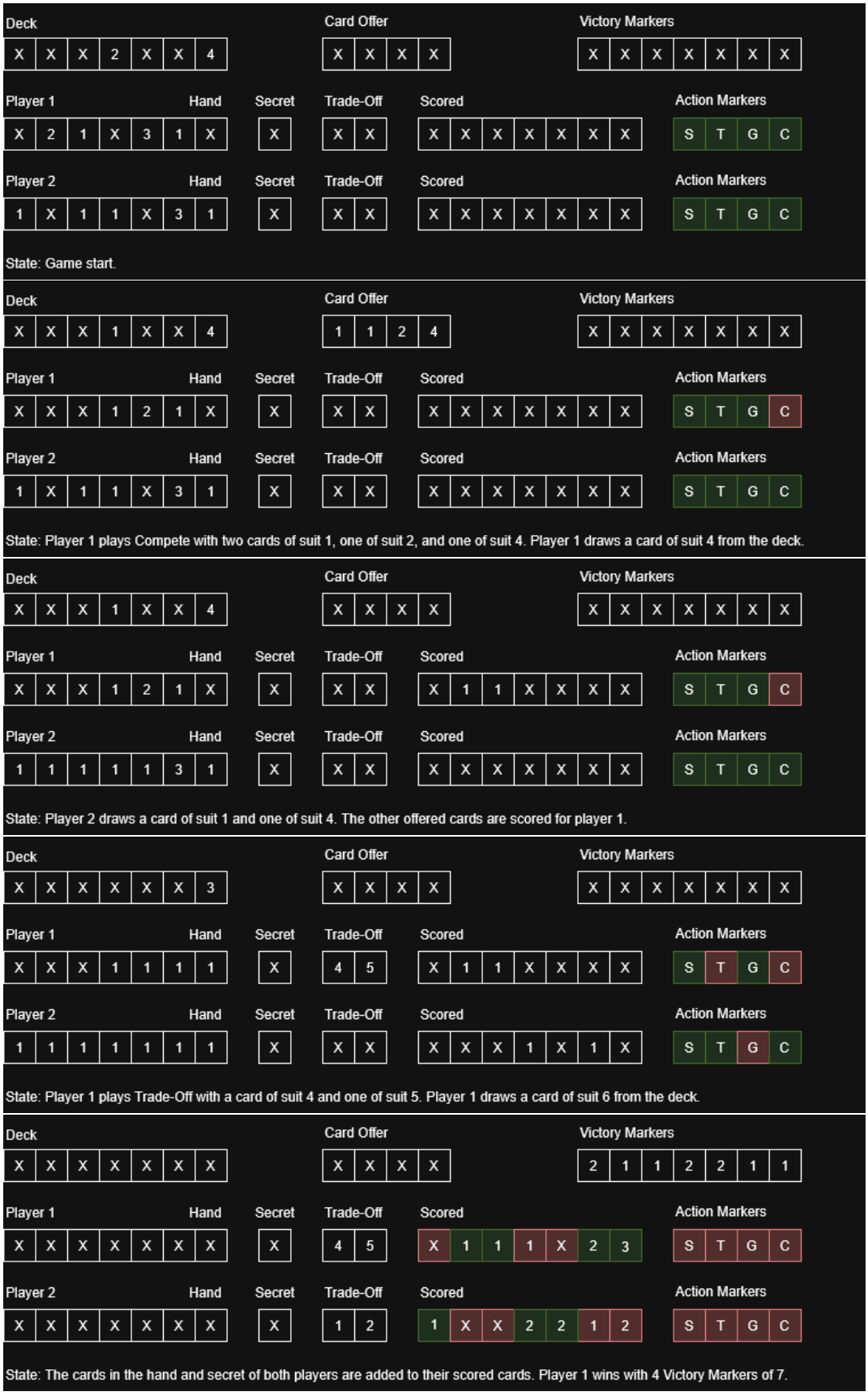

Figure 2: Examples of moves and game outcomes.

4 Game States

In order to represent games from start to finish for the agents, game states must be stored in a manner that enables them to traverse through, copy, and compare them. These states include the following properties:

1. The Victory Markers
2. The Action Markers
3. The players' secret and discarded cards (if they have them)
4. The players' current hands
5. The cards in the deck
6. What turn and round it is
7. The next action to be taken (playing or drawing cards) and which player must take it
8. The cards being offered to a player in a Gift or Compete (if applicable)

Between any given action, all but the first two change, and the nodes that mark the start of a new round will change those two as well. The states at the end of a game also have a result value $a \in \{-1, 0, 1\}$ which indicates a victory by the second player, a draw, and a victory by the first player respectively. We will henceforth denote a game state with $G_{t,r}$, where $t \in \{1, 2, 3, 4\}$ represents what action or turn of the round it is, and $r \in \{1, 2, 3, T\}$ what round of the game it is, with T denoting a tiebreaker.

The states must also be partially randomized when the agent interacts with them, to ensure that the agent acts with incomplete information. Some experiments are performed without this randomization, to analyze how the performance of an agent improves when it is given the ability to "cheat" when choosing its move.

4.1 Randomizing States

When an agent is performing rollouts without cheating, it must randomize all parts of the board it cannot see. These include the opponent's hand, secret and discarded cards, and the deck. The scored cards, Victory and Action Markers, and the agent's own cards are not randomized, as those are always visible to a real player of the game. To accomplish this randomization, the opponent's unscored cards are all returned to the deck. An equivalent number of cards is then dealt back to the opponent, who then selects random cards for Secret and Trade-Off if applicable. An example of this randomization can be seen in the figure on the following page.

This randomization keeps the agent's information incomplete, and the impact thereof will be examined in the experiments. The randomization is done before every rollout, to ensure the player's evaluation is not excessively influenced by one particular randomized state where one move may be more desirable as opposed to another. When the agent is offered cards, the randomization also does not affect the cards they are offered, as the offered cards are all visible to a real player. The randomization does, however, affect cards the agent has offered the opponent on previous turns.

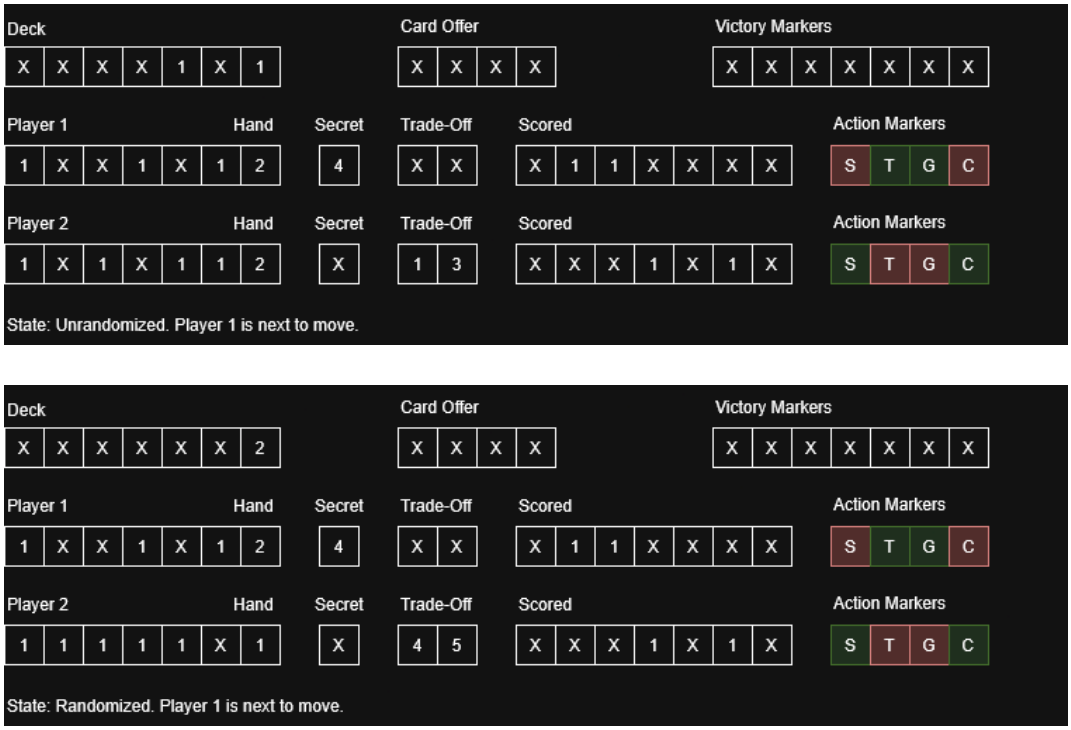

Figure 3: Player 1 performs a rollout with player 2's state being randomized.

5 Types Of Agents

Several kinds of agents have been programmed to emulate real players. The simplest one is one that chooses its moves entirely randomly (henceforth referred to as the **random agent**), with its only constraint being that it must play legal moves. The following agents all use the Monte Carlo System, either exclusively or in conjunction with other algorithms for decision-making. The underlying logic of each agent will be explained in the remainder of the section.

5.1 Simple Agent

The simple agent is the most rudimentary implementation of the Monte Carlo system for this game and serves as a good foundation for the more complex agents. It works by making a copy of the game state for every potential move it can make, performing n rollouts consisting of letting random players finish the game from said states, and then evaluating the average outcome of every state, selecting the move that led to the most desirable result. This agent does not look further ahead than a single turn in the game.

The quality of these results will be determined by the **promise value** $\mu(a) \in [-1, 1]$. Depending on the player, the state will be selected with either the highest or lowest value of $\mu(a)$, which is calculated as the mean of game outcomes a collected over n rollouts. The complete logic can be seen in the pseudocode of Algorithm 1 on the following page.

Algorithm 1 simple agent Move Evaluation

```
1: stateList := [ ]
2: for action in actions not yet taken do
3:     for combination in card combinations in hand do
4:         cloneState := clone(currentState)
5:         makeMove(action, combination, cloneState)
6:         stateList ← stateList + cloneState
7:
8: promiseValues := [ ]
9: for i = 1 to size(stateList) do
10:    nextPromiseValue := 0
11:    bestState := 0
12:    for j = 1 to sampleSize do
13:        sampleOutcome := result(playrandomRound(stateList[i]))
14:        nextPromiseValue ← nextPromiseValue + sampleOutcome
15:    promiseValues ← promiseValues + nextPromiseValue / sampleSize
16:
17: if playerID = 1 then
18:    return stateList[max(promiseValues)]
19: return stateList[min(promiseValues)]
```

5.2 Tree Search Agent

The tree search agent, or the MCTS agent, utilizes MCTS to look further ahead than a single turn. It composes a tree of states out of nodes consisting of five properties.

Game State

A node contains a state $G_{t,r}$ where r must always be equal to the value of its parent, as move evaluations cannot extend between rounds, naturally limiting the potential size of a search tree as the agents near the end of a round. the value of t may either be the same as that of its parents, in the case of a state not describing the final action in a turn, or one greater than the parent otherwise. The root contains the present state of the game itself. States are only randomized within leaf nodes if randomization is enabled. This does allow the agent to cheat to an extent, as all states above the leaves are reachable states from the root. This is however necessary as randomizing internal states can expand the tree with states that cannot be reached from the root.

State Transition Descriptor

A node contains a notation of the action that created its own state out of the state of its parent. These actions exist within three categories:

1. **Agent moves**: These are any moves performed by the agent currently constructing a state tree, which can be either an action taken on the agent's own turn or a choice of cards to draw from a Gift or Compete by the opponent.
2. **Adversary Moves**: These are any moves performed by the opponent. These include the opponent's moves and selections from the agent's own Gift or Compete.
3. **Random Moves**: The only move in this category is drawing a card from the deck at the end of either player's turn, a move that is controlled by a separate random number generator.

The state transition descriptor of a node determines the states of its children, as well as the node type. In the root, this descriptor is set to a null value.

Node type

A node, corresponding to the three types of state transition descriptors, can be marked as one of three types. The chosen type corresponds to the state transition descriptors of the children of the node. In the root, these will always be agent moves.

1. **Max node**: The highest evaluated child is selected in the final evaluation. The state transition descriptors of the children must be agent moves. In other words, the agent is next to move in the state being evaluated.
2. **Min node**: The lowest evaluated child is selected in the final evaluation. Functions identically to a max node with adversary moves and the opponent being next to move.
3. **Chance node**: The mean value of all children is selected in the final evaluation. All children will have random moves as state transition descriptors.

Since leaves do not have any designated type, their final value is determined with Algorithm 1.

Promise Value

After the leaves are all assigned a promise value $\mu(a(G_{t,r}))$, these values are propagated back to the root corresponding to the types and descriptors mentioned prior. After this, the agent chooses the state with the best $\mu(a)$ in the children of the root, performing the move that aligns with its state transition descriptor. These properties enable a bottom-up implementation of the Minimax algorithm [DM].

Depth

The root's depth d_{root} is set to the depth d of the agent. The depth d_2 of any node is set to $d_1 - 1$ with d_1 denoting the depth of its parent. All leaves must be of depth $d < 1$. The agent may extend subtrees to negative d values in certain conditions to ensure that every leaf is a chance node. This is done to ensure Algorithm 1 works correctly and in practice makes certain depths work identically. This will be relevant in Experiment 8.7.

A top-down implementation of this algorithm is also possible, and would enable $\alpha - \beta$ pruning [KM75], likely on the basis of an Upper Confidence Bound [CM07], but due to constraints in the simulator, a bottom-up method was chosen instead. The difference between the approaches is beyond the scope of this research.

5.3 Recursive Simple Agent

While the simple agent uses random agents in its rollouts, a possible method to perform searches of increased depth without the use of MCTS is to let a simple agent perform rollouts with other simple agents. These will be labeled recursive rollouts n_1 with simple agents with rollouts n_2. The recursive method can more closely imitate a top-down implementation of a minimax algorithm. However, a recursive rollout consumes more runtime as it performs regular rollouts at every child of a state. The nature of this growth in runtime potentially reveals information about the game's state space, which will be examined in greater detail in Experiments 8.7 and 8.8.

6 Pre-Sampled States

As the game state resets almost entirely between rounds, one method to reduce the length of rollouts is to pre-sample all states $\{G_{4,t} \mid t > 1\}$. The only aspect of a state persistent between rounds is the Victory Markers, which can have non-zero values only after the first round. The amount of number arrangements can be mathematically deduced as 3^7, as there are 7 Victory Markers, each with 3 possible states: claimed by player 1, claimed by player 2, and unclaimed. Among these arrangements, many will have decisive outcomes that do not continue the game to another round or tiebreaker. These arrangements are consequently not given a value $\mu(a(G_{4,r}))$. We will store the values of these arrangements in two ways.

Array

Since the arrangement of the Victory Markers, denoted by $(m_1, \ldots, m_7)$, can be applied as an integer in base-3, we can express the arrangement in base-10 as we index it into an array with the following formula:

$$\sum_{i=1}^{7} m_i * 3^{i-1}$$

The use of an array could increase search times in memory. We can curtail this potential issue by storing the values in a different structure.

Tree

Every Victory Marker can be mapped to a layer of ternary nodes in a tree, where each branch of a node at depth i represents the value of the Marker m_i. By traversing the according branches to a leaf containing the promise value corresponding to the combination of Victory Markers in the current game state, we can reduce the theoretical search time to $3 * 7$ steps. Whether this impact is noticeable is a minor point of interest in the experiments.

A pre-sampled rollout itself will only continue to the end of the round it started in, rather than to the natural end of the game. It subsequently returns the sampled value of the attained combination of Victory Markers as a. The difference in efficiency between a pre-sampled and regular will be analyzed in detail in the experiments, as well as the results contained within the pre-sampled database themselves.

7 Agent Configuration

We will now outline the full list of adjustable parameters for an agent as they will be applied in the experiment:

- **Rollouts**: The number of games an agent will simulate to evaluate the likelihood of victory from a state. A random agent will be reduced to a simple agent with 0 rollouts. Denoted by n. In the case of the recursive simple agent, this is split into n_1 and n_2.
- **Depth**: The depth of a tree of states being searched by the agent in its evaluations. A simple agent will be reduced to a tree search agent with a depth of 1. Denoted by d.
- **Cheating**: A cheating agent performs rollouts without randomizing the game first, granting it complete information about the game state.
- **Pre-sampling**: A pre-sampling agent will not require rollouts to go to the end of the game. Instead, it will play them to the end of the round, and then compare the end state to a pre-sampled list of values for its evaluation.

8 Experiments

8.1 Simple Agents vs. Random Agents

In this experiment, we let random agents play against simple agents with various rollout counts.

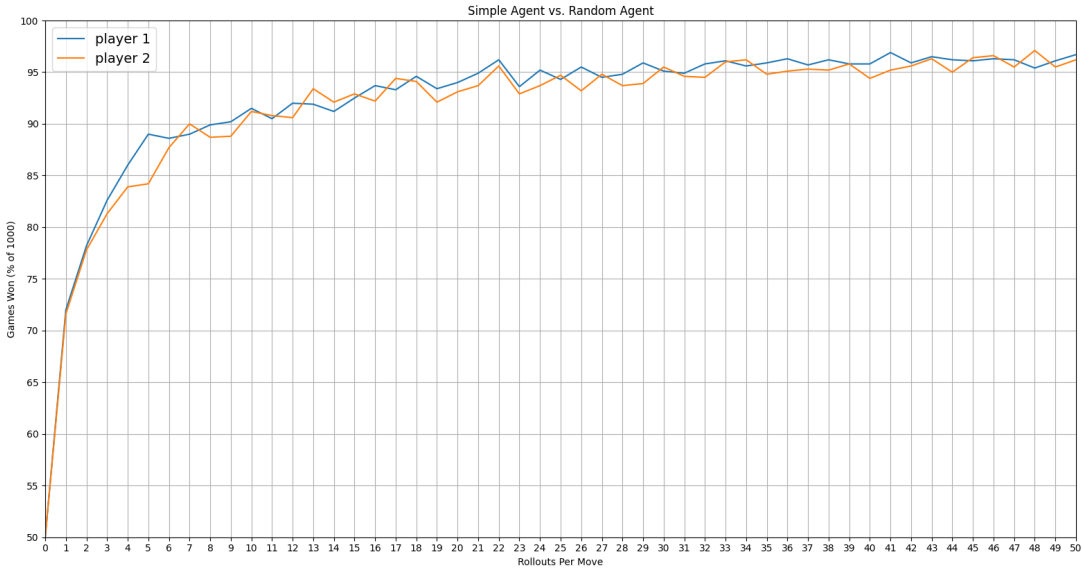

Figure 4: Performance of the simple agent against random agents.

Figure 4 shows the performance of the simple agent when it is afforded incomplete information in its rollouts. We can observe that as we increase n, the win rate (denoted by W_p) improves asymptotically. The upper limit in this case is approximately 96%. Note that the graph begins fluctuating around this limit at fairly low rollout counts, progressively stabilizing near said limit as the count grows. The simple agent with 20 rollouts will serve as our baseline performance benchmark in subsequent experiments, as it already converges upon the perceived limit while being far more efficient in runtime, seen in the table below.

n	W_1	W_2	Runtime (s)
1	72.0%	71.6%	1.10
10	91.5%	91.2%	3.59
20	94.0%	93.1%	6.14
50	96.7%	96.2%	14.37

Table 1: Success rate and runtime for 1000 games against random agents

Note that these runtimes were achieved by agents without any ancillary evaluation techniques. The next experiment will concern the efficacy of one such technique. Namely, pre-sampling game states for future evaluations.

8.2 Pre-Sampling Trends

In the following experiments, cheating simple agents with 50 rollouts per turn played 100 games from the following states:

$$\{G_{t,r} \mid t = 1 \wedge r \in \{2, 3\}\} \text{ for all } M \in \{(m_1, \ldots, m_7) \mid m_i \in \{0, 1, 2\}\}$$

This represents all possible combinations of Victory Markers on the second and third rounds. $\mu(a)$ was gathered from 100 games for every combination. The results will serve as the pre-sampled data for agents in subsequent experiments. In the graph below, the bars represent the mean of $\mu(a)$ where either player 1 or 2 held a specific Victory Marker at the start of the round. For player 2, the signs of these results are flipped for simplified visualization. Furthermore, any state with a decisive outcome (e.g.: $M = \{1, 2, 1, 1, 2, 1, 2\}$) was excluded from evaluation to avoid redundancy.

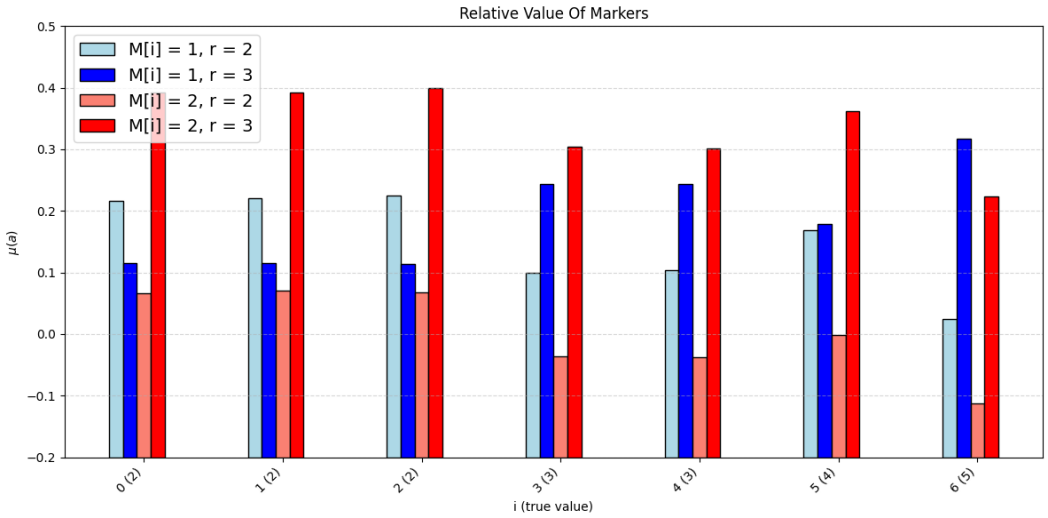

Figure 5: Average outcome P respective to each Marker

The most prominent observation is how differently the agents evaluate states as a whole depending on what round it is, with player 1 favoring round 2 and player 2 favoring round 3. This indicates a potential disadvantage attributed to moving first. Additionally, Markers with an even number of cards in the deck are generally valued higher than ones with an odd number. This is potentially due to those Markers being easier to keep once claimed, as only half the cards of the respective suit need to be scored. Another noteworthy trend is that the perceived value of Markers generally becomes lower the higher their real value.

Player 1 notably deviates from these trends in round 3, attributing higher values to Markers with odd-numbered real values, and higher values in general. This is presumably a result of tie-breaker outcomes, where Charm Points will most often determine the victor. This implies that playing to reach a tie-breaker may be an effective strategy for player 1 if the game reaches the third round.

The final trend worth noting is the generally more extreme evaluations attributed to games in round 3. This is likely a result of the aforementioned turn order advantage as well as the potential viability of playing for tie-breakers. With these values immediately accessible, the next experiment will have the agents apply them to their rollouts.

8.3 Impact Of Pre-Sampling

In this experiment, simple agents of 1-50 rollouts were tested against simple agents of 20 rollouts in three different conditions: One where neither apply pre-sampling in their rollouts, another where only the tested agent applies pre-sampling to its rollouts, and a final one where both utilize pre-sampling simultaneously. The main point of interest is runtime improvement. Potential changes in accuracy will also be analyzed.

Runtime Efficiency

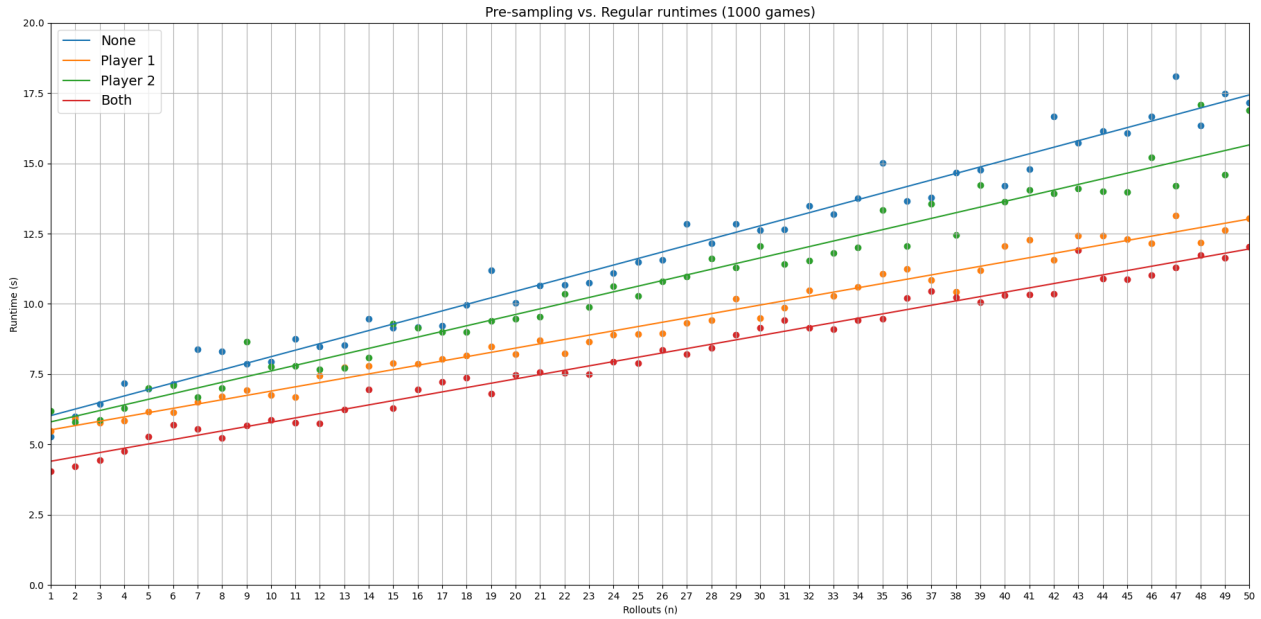

Figure 6: Average runtimes across scenarios

In all scenarios, the runtime grows in a roughly linear trend. The most noteworthy observation is the difference in growth rates when only the tested agent applies pre-sampling, with the adversary sampling as normal. We see that player 1 experiences a substantially larger decrease in runtime than player 2 and that this benefit scales with the rollout count. This is potentially related to tie-breakers and the average game length of games won by player 1 as opposed to player 2, an implication further reinforced by the results of Experiment 8.2. Regardless, the benefits in runtime are sufficiently desirable for both players.

The impact of storing the tables in a tree as opposed to an array was negligible, likely due to the tables being too small for the search time to be significant. This could also be the result of hardware-related behaviors such as values being cached and the associated cost of memory reads. The searches through the pre-sampled data will use the array structure exclusively in subsequent experiments.

Influence on Accuracy

We can measure this by the quotient between the mean win rate of the tested agents, dividing the mean outcome of pre-sampled games by the mean outcome of regularly sampled games.

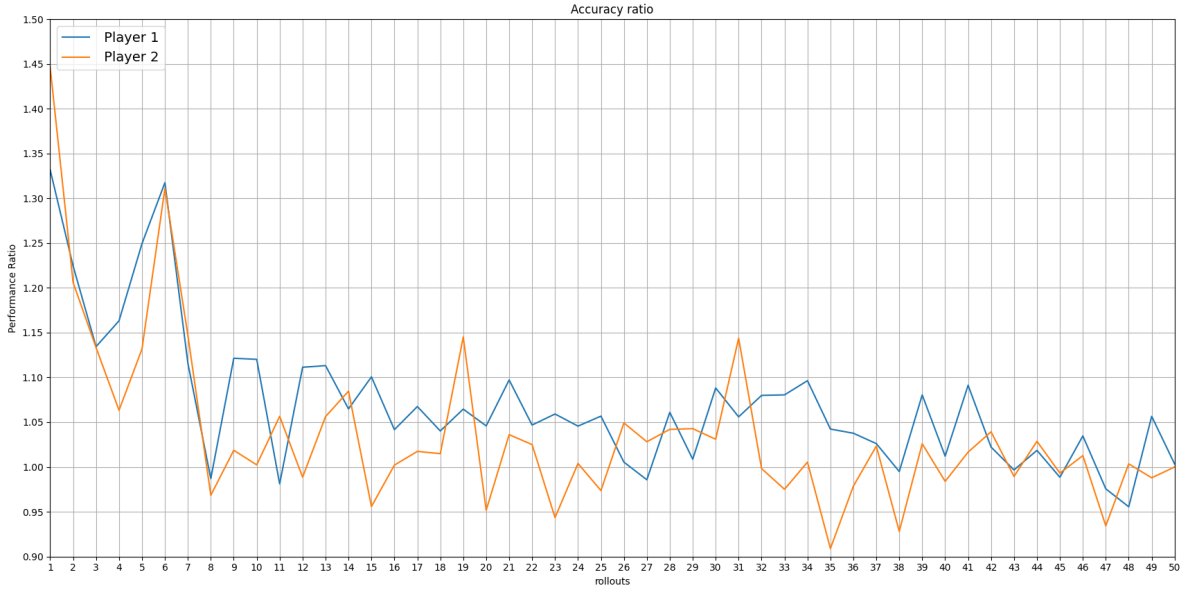

Figure 7: Accuracy ratio between pre-sampling and regular sampling

The fluctuating ratio indicates that any perceived increase in accuracy is likely quite tenuous. However, there are still observations worth mentioning. Firstly, at the lowest rollout counts, the ratio is substantially higher. This is likely a result of a single pre-sampled rollout providing more information than a regular rollout. Secondly, as the rollout count increases, the ratio visibly converges towards 1. This is ostensibly due to the information gained from each rollout being increasingly less important as more are performed, and the best moves become increasingly clear. Finally, player 1's ratio is markedly more consistent and generally higher than that of player 2. Player 2's accuracy when pre-sampling still remains consistently on par with its regular accuracy, rarely falling below 0.95. In subsequent experiments, the agents will apply pre-sampling unless otherwise specified, with the implicit expectation that the information benefits player 1 the most.

8.4 Impact Of Incomplete Information

In this experiment, a simple agent of 20 non-randomized rollouts, which we will henceforth refer to as a cheater, was tested against agents of increasing rollouts that remained randomized.

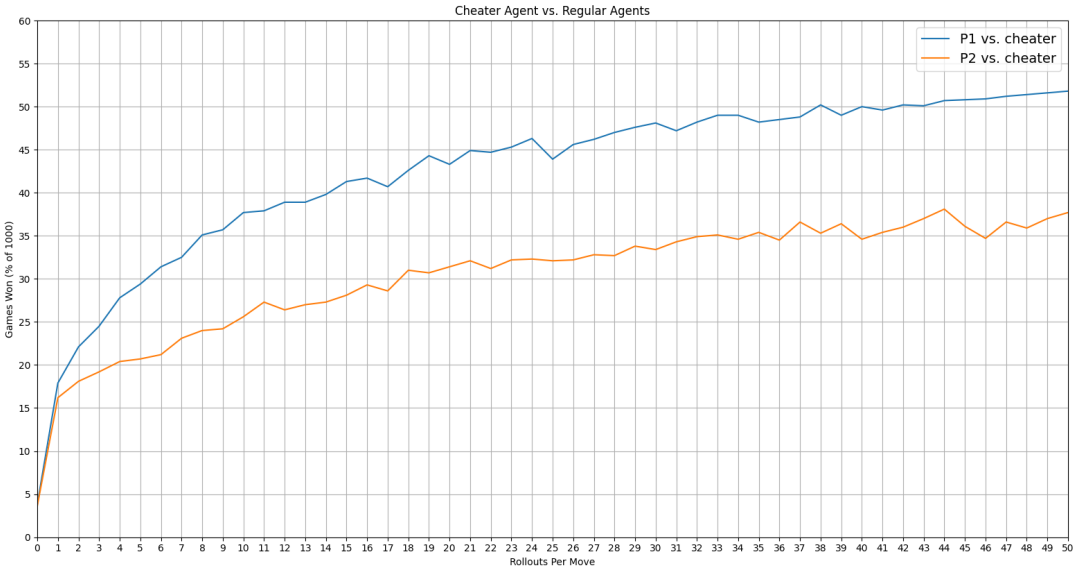

Figure 8: Performance of regular agents against a cheating agent.

n	50	60	70	80	90	100
W_1	47.8%	51.1%	51.1%	52.8%	53.4%	54.0%
W_2	43.7%	45.1%	47.5%	47.3%	49.6%	49.0%

Table 2: Success Rates tested with higher rollout counts and no pre-sampling.

Beyond the expected result of the cheater outperforming any adversary with a rollout count not substantially larger than its own, one incipient overarching trend can be observed within both this and prior experiments. As the information gained within an individual rollout becomes more complete, player 1 outperforms player 2 by an increasingly wide margin. Note that player 2 ostensibly needs over 5 times the rollout count of player 1 to achieve a win rate of 50% without cheating (see Table 2), a ratio which is likely further amplified by pre-sampling. By contrast, player 1 only needs roughly double the rollouts to achieve the same result with pre-sampling, and 3 times without it.

Another notable observation is a potential limit in win rate achievable against the cheater by a regular agent, illustrated by the diminishing increase in win rate. This shows that a non-randomized rollout is vastly more informative than a regular one, and that optimal play can be achieved with them far more easily. In upcoming experiments, agents will not cheat unless otherwise specified.

8.5 Rounds And Game Length

Now we will both observe results from prior experiments and perform trials to test the validity of our observations thus far, to extrapolate potential correlations between game lengths and player advantages. Firstly we will observe the dataset from the previous experiment for the average length of a game won by each respective agent:

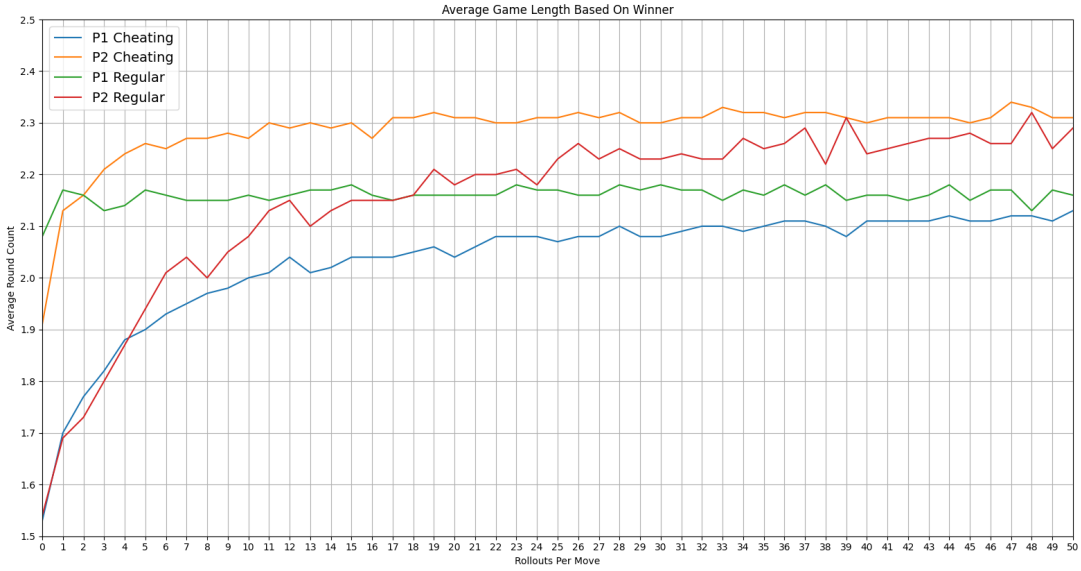

Figure 9: Mean round counts of games between a cheater and a regular agent.

We can immediately observe the points of convergence at the end of the graph, being approximately 2.15 rounds for player 1 and 2.3 rounds for player 2. Another prominent trend is player 1's propensity to play shorter games when it is the more optimal player, a trend not observed within player 2. This further reinforces the assumption that player 1 makes an active effort to win before round 3, while player 2 does not. Below are the results of trials between agents of equal rollouts that are both cheating and pre-sampling. Note that $n = 0$ corresponds to a random agent. The following table shows how the percentage of games that went to round 3 that were decided by tiebreaker, denoted as TB, as well as the win rates of player 1 and 2 in said games.

n	0	10	20	30	40	50	60	70	80	90	100
TB	2.38%	3.25%	3.35%	4.21%	4.62%	4.73%	4.51%	5.26%	5.06%	5.65%	4.85%
$W_1(T)$	41.7%	53.2%	37.0%	38.9%	42.0%	42.9%	33.8%	46.3%	38.0%	43.9%	31.9%
$W_2(T)$	58.3%	46.8%	63.0%	61.1%	58.0%	57.1%	66.2%	53.7%	62.0%	56.1%	68.1%

Table 3: Tiebreaker statistics

The table on the following page shows, for each agent, what percentage of their total wins took place in each round, with $\mu(l)$ denoting the mean length of their victories in rounds.

n	$\mu(l)$	$W_1(1)$	$W_1(2)$	$W_1(3)$	$W_1(T)$
0	1.93	27.2%	52.4%	20.0%	0.4%
10	2.11	8.4%	72.0%	18.7%	0.9%
20	2.16	5.4%	73.6%	20.3%	0.7%
30	2.17	5.1%	72.6%	21.4%	1.0%
40	2.18	5.1%	72.4%	21.4%	1.2%
50	2.16	5.1%	73.5%	20.1%	1.2%
60	2.16	5.0%	73.7%	20.4%	0.9%
70	2.18	4.6%	73.2%	20.7%	1.5%
80	2.17	4.8%	73.3%	20.8%	1.2%
90	2.19	5.2%	70.8%	22.4%	1.6%
100	2.18	5.1%	71.6%	22.1%	1.2%

Table 4: Results of player 1

n	$\mu(l)$	$W_2(1)$	$W_2(2)$	$W_2(3)$	$W_2(T)$
0	1.92	27.6%	52.5%	19.4%	0.6%
10	2.28	13.8%	44.3%	40.9%	1.1%
20	2.38	10.5%	40.9%	47.0%	1.7%
30	2.42	8.8%	40.5%	48.6%	2.1%
40	2.44	8.8%	38.0%	50.9%	2.3%
50	2.48	8.1%	36.2%	53.4%	2.3%
60	2.47	8.6%	36.0%	52.8%	2.5%
70	2.50	7.5%	34.8%	55.1%	2.6%
80	2.50	7.7%	34.6%	54.9%	2.8%
90	2.51	7.2%	35.0%	54.9%	2.9%
100	2.52	6.3%	35.3%	54.7%	3.8%

Table 5: Results of player 2

In Table 3 we can observe that as the rollout counts rise, the rate of tiebreakers does as well. This growth does not appear to diminish within the tested range either. We can also see that player 2 is increasingly favored by tiebreakers as more rollouts are granted, disproving our hypothesis from Experiment 8.2. However, the impact of tiebreakers remains resoundingly minimal, likely to player 2's overall detriment. In Tables 4 and 5 we can see that in round 1, player 2 is consistently favored. However, the overall decline in both $W_1(1)$ and $\mu(l)$ shows that more optimal agents will less frequently end the game in the first round. We can see that player 1 is quite vastly favored in round 2, presumably by virtue of them moving last. Round 3 also consistently favors player 2, and as fewer games end in round 1, player 2's victories become more frequent in round 3.
More generally, we can see that $\mu(l)$ rises even further than initially observed for player 2 in Figure 9, up to approximately 2.5 rounds on average, while player 1 remains stable at 2.15–2.2 rounds on average. Based on these results we can infer the following:

- Most games end in round 2. This is evidenced by the fact that even random agents win most of their games in round 2 and that their $\mu(l)$ is approximately 1.9. It also explains why player 1, whose $\mu(l)$ consistently remains closer to 2, outperforms player 2 in Experiment 8.4.

- As the agents become more optimal in their evaluation, player 1 will outperform player 2 by an increasingly broad margin. This is implied by $\mu(l)$ growing only for player 2, and $W_1(1)$ being far higher than $W_2(1)$.

- There is an inherent advantage attributed to being the last to move in a round. Not only is this further reinforced by results from Experiment 8.2, but it also aligns with player 2's growing values of $\mu(l)$ and $W_2(3)$. This implies that as fewer games end in round 1 and player 2 wins less frequently in round 2 where it moves first, it ostensibly more actively attempts to force games to reach round 3, where it may once again be the last to move.

- While player 2 is consistently favored by tiebreakers, it cannot force them frequently enough for this to be relevant to their success, further emphasizing player 1's advantage that grows in tandem with increasingly informed and extensive move evaluations.

8.6 Preferred And Forced Actions

In the following experiment, we observe what moves the agents tend to prefer within a given turn of a round, and how their performance changes should they be forced to deviate from said preference. We will apply the notation of S, T, G, C for Secret, Trade-Off, Gift and Compete respectively. To start, we will let two standard simple agents with 50 rollouts play a large number of regular games, and aggregate the move distribution on each turn:

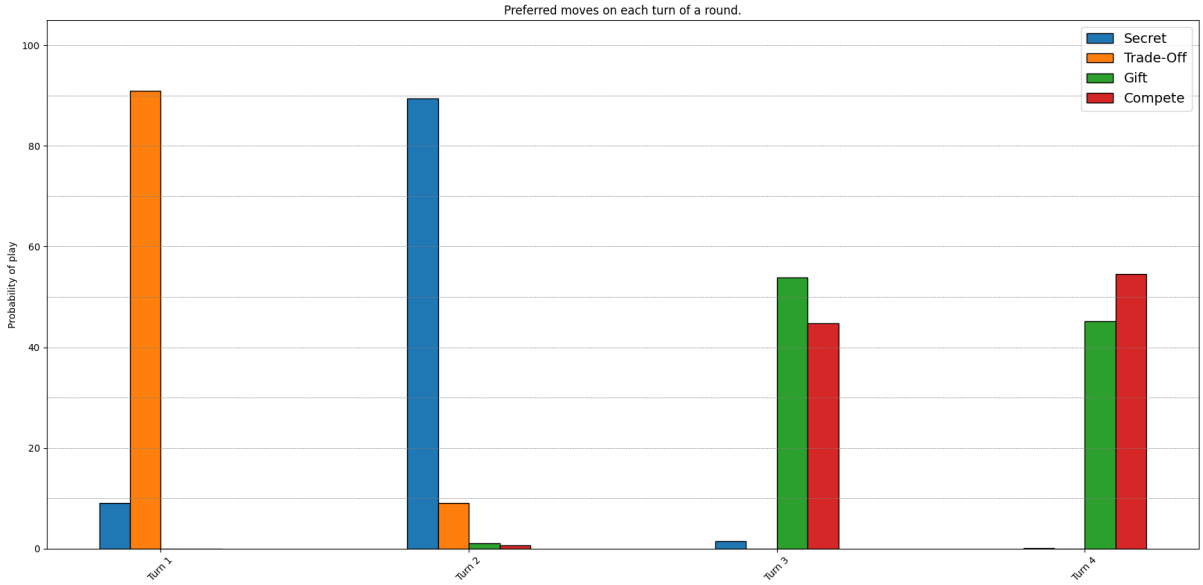

Figure 10: Most common moves on each turn.

The agents show an overwhelming preference for Trade-Off on the first turn, and Secret on the second. They will choose to Gift on turn 3 marginally more often than Compete, though the disparity is far less severe. This implies that moves that offer cards to the opponent are evaluated lower in earlier turns. This could be due to various reasons, as these actions reveal cards to the opponent, reduce the number of available moves on future turns by virtue of taking more cards to play and give the opponent more immediate influence on the agent's hand. By comparison, Secret and Trade-Off seem to be regarded as less risky and more consistent early on. This does not inherently confirm the optimality of this move order, it only shows that the simple agent's logic favors moves that do not offer cards.

With this precedent set, we will now test the agents in trials where one agent must play a set move order every round, and the other does not. We will do this for every possible move order, and evaluate which move order has the largest impact on performance. The agents are pre-sampling, not cheating, and using 50 rollouts as they did before. The moves are ordered with the notation mentioned previously.

Order	W_1	W_2	$\mu(l \mid a = 1)$	$\mu(l \mid a = -1)$
S-T-G-C	58.0%	45.9%	2.10	2.18
S-T-C-G	56.0%	48.4%	2.16	2.22
S-G-T-C	52.7%	41.4%	2.13	2.07
S-G-C-T	57.6%	54.8%	2.13	1.55
S-C-T-G	47.9%	41.4%	2.19	2.03
S-C-G-T	57.5%	57.2%	2.16	1.53
T-S-G-C	55.5%	41.9%	2.13	2.19
T-S-C-G	54.8%	42.7%	2.14	2.25
T-G-S-C	58.7%	44.3%	2.11	2.21
T-G-C-S	50.4%	42.3%	2.15	2.15
T-C-S-G	56.5%	44.4%	2.16	2.18
T-C-G-S	55.6%	47.1%	2.14	2.15
G-S-T-C	54.3%	42.3%	2.14	2.09
G-S-C-T	56.7%	55.2%	2.15	1.56
G-T-S-C	53.6%	41.9%	2.14	2.05
G-T-C-S	47.4%	40.1%	2.15	2.05
G-C-S-T	53.2%	57.9%	2.15	1.44
G-C-T-S	33.7%	29.5%	2.18	1.79
C-S-T-G	48.5%	39.7%	2.18	2.02
C-S-G-T	55.2%	54.8%	2.14	1.54
C-T-S-G	46.9%	36.9%	2.18	1.96
C-T-G-S	45.0%	39.8%	2.18	1.96
C-G-S-T	54.0%	59.2%	2.15	1.46
C-G-T-S	33.5%	29.9%	2.19	1.75

Table 6: Results achieved by different move orders

The most striking trend is the fluctuation of $\mu(l \mid a = -1)$ based on move order. Playing Trade-Off last consistently improves player 2's performance, leading to the shortest wins on average. Player 1 is markedly less volatile, with $\mu(l \mid a = 1)$ stabilizing at approximately 2.1–2.2 and the win rate remaining above 50% in the majority of cases. Also worth noting is that the two most severely disadvantageous move orders for both players involve playing Trade-Off third and Secret last. The perceived ideal plays on turns 2 and 3 are rather vague and depend on the moves chosen in turns 1 and 4. In turn 1, Secret is a very consistent and safe first move for both players, Trade-Off is only desirable for player 1, Gift can be effective provided Secret is not played last, and Compete is overall the least consistent, though player 2 performs better when it offers cards on its first 2 turns and does not play Secret last. In turn 4, the best move is Trade-Off by a very wide margin, followed by Compete, followed by Gift, with Secret being decidedly the worst.
Overall, this shows that the simple agent's evaluation is quite effective for player 1, but flawed for player 2, as when this hard-coded move order was used, player 2 performed better than with the preferred move order it evaluates naturally. The fluctuations in $\mu(l \mid a = -1)$ imply that these move orders are more conducive to first-round wins which greatly benefit player 2. We will not use hard-coded move evaluation constraints or heuristics in upcoming experiments.

8.7 Monte Carlo Tree Search

In the following experiment, we tested agents using the MCTS algorithm with pre-sampled rollouts against various agents. We will first assess whether their performance and playing patterns diverge from the simple agents in any significant way. For the entire experiment, the MCTS agent was player 1. Note that an MCTS agent with a depth of 1 is equivalent to a simple agent. In this experiment, we will be labeling agents by depth with $C(d)$ if they are cheating, and $H(d)$ otherwise. Due to hardware constraints, aggregating performance results for depths above 3 is not possible.

Playing vs Random Agents

d	$H(1)$	$H(2)$	$H(3)$	$C(1)$	$C(2)$	$C(3)$
W_1	93.6%	96.7%	93.2%	95.4%	98.3%	95.6%
$\mu(l \mid a = 1)$	1.55	1.42	1.63	1.53	1.44	1.65

Table 7: MCTS agents ($n = 20$) against random agents (1000 games)

The most key observation is that increasing depth beyond 2 appears detrimental to performance, as while W_1 remains similar between depths 1 and 3, the average round length rises by approximately 0.1. Additionally, the MCTS agent with depth 2 even outperforms the simple agent from Experiment 8.1, implying some benefit to one added layer of search depth. With this precedent set, we will now run the same trials again, but against a simple agent with 20 pre-sampled rollouts.

Playing vs Simple Agents

d	$H(1)$	$H(2)$	$H(3)$	$C(1)$	$C(2)$	$C(3)$
W_1	53.6%	31.1%	29.9%	68.4%	44.7%	33.6%
$\mu(l \mid a = 1)$	2.06	1.97	2.07	1.53	1.83	1.96

Table 8: MCTS agents ($n = 20$) against simple agents ($n = 20$) (1000 games)

These results contradict the hypothesis that an extra layer of search depth is beneficial but align with the overarching trend that as an agent becomes more optimal as player 1, it will make an active effort to keep games shorter. The MCTS agent appears to perform far more poorly in longer games, see the table below:

Win rate	$W_1(1)$	$W_1(2)$	$W_1(3)$	$W_1(T)$
$H(1)$	46.3%	60.4%	42.5%	62.5%
$H(2)$	16.3%	42.3%	19.7%	50.0%
$H(3)$	12.7%	39.7%	2.5%	25.0%
$C(1)$	59.7%	77.6%	50.8%	0.0%
$C(2)$	31.7%	58.6%	21.9%	0.0%
$C(3)$	21.6%	45.4%	15.8%	0.0%

Table 9: MCTS agents vs. simple agents

This shows that MCTS agents generally follow the same trends in game length and result as simple agents. We will now observe the move order preference of an MCTS agent with a depth of 2 in the plot below:

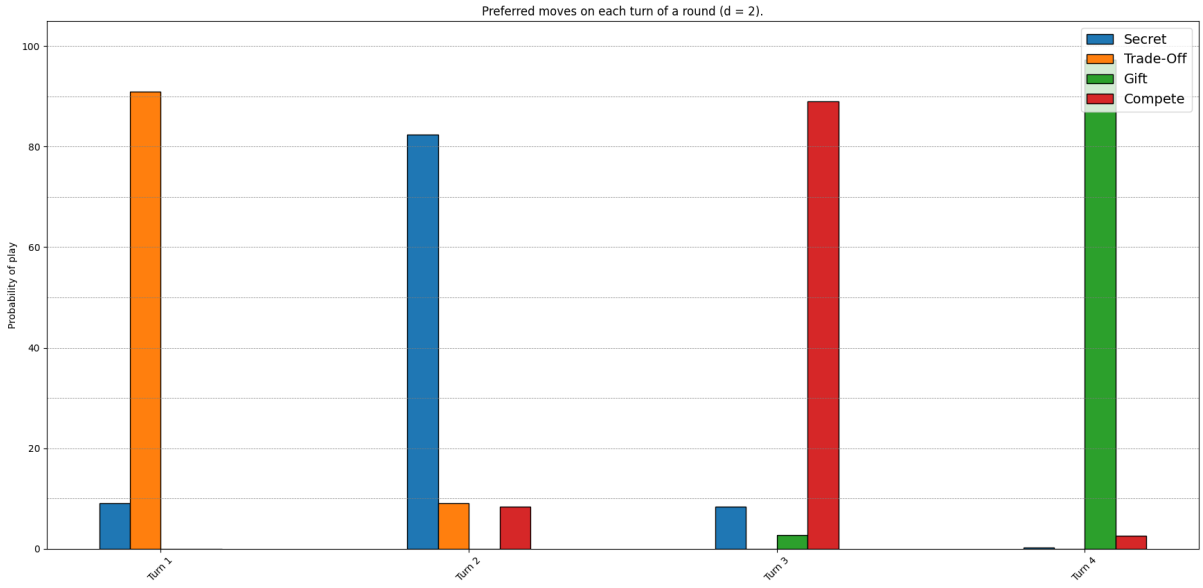

Figure 11: Most common moves on each turn ($d = 2$).

Similarly to the simple agent, a strong preference for Trade-Off first and Secret thereafter can be observed. Unlike the simple agent, however, it also very strongly prefers Compete on the third turn, whereas the simple agent only slightly preferred Gift on the third turn. In Table 6 we see that the two orders make a negligible difference for both players. Furthermore, the MCTS agent will sometimes play Compete on the second turn and Secret on the third. Although this move order was shown to be marginally more effective than the naturally preferred orders, it does not appear to significantly improve the MCTS agent's performance. This could likely be a result of the topic of the next section.

Search Tree Size

In a round, an agent will perform 6 actions. It will make 4 moves of its own, and take cards offered by the opponent into its own hand in their Gift and Compete actions. This means up to 18 actions are taken in a game. The player will draw three cards from the deck over the course of each round. This means the player's state changes 9 times every round, and the game state is changed 18 times per round, 9 times by each player, and thus 54 in total.

Because we are not able to extend a search tree across multiple rounds, the theoretical maximum depth is 18 for any given search tree. This is relatively shallow compared to other games where MCTS is a viable algorithm. However, the combinations of possible cards and moves in a turn are far larger, and this may lead to move evaluation with MCTS being overly complicated and

possibly detrimental overall. In the following table, we can see the average size of a search tree on action x of round r in state nodes. Note that due to the behaviors outlined in Section 5.2, the agent functions identically on $d = 3$ and $d = 4$ Additionally, the sample sizes are rather small due to hardware constraints, making the results less accurate.

r, x	$d = 1$	$d = 2$	$d = 3$	$d = 5$
1, 1	52.9	227.5	14138.4	587233.4
1, 2	31.3	101.1	2301.6	33934.0
1, 3	19.6	46.7	189.9	245.8
1, 4	4.8	10.2	15.8	17.3
1, 5	5.4	4.7	5.7	5.0
1, 6	4.6	4.5	4.6	4.6
2, 1	52.8	205.5	5256.4	118195.9
2, 2	29.0	79.1	563.8	5446.5
2, 3	4.3	10.7	140.7	137.0
2, 4	37.6	72.5	21.8	32.2
2, 5	4.7	4.7	4.4	4.3
2, 6	12.8	10.0	7.6	7.8
3, 1	51.6	232.5	13439.6	548100.1
3, 2	29.9	91.9	1691.3	26289.0
3, 3	19.1	42.0	169.8	236.2
3, 4	5.2	10.0	8.9	20.9
3, 5	6.0	5.2	5.7	4.8
3, 6	4.4	4.1	4.5	5.0

Table 10: Average search tree size per action across 100 games

The main point of interest is the sizes of trees in the first two actions of every round. We can observe that they grow exponentially with every added layer. As the round progresses, both players will have consumed more Action Markers and will have fewer cards in their hand, narrowing the search tree immensely. This, in conjunction with the results of the prior experiments, implies that larger search trees hinder the accuracy of move evaluations, as while the chosen move orders appear correct, the cards chosen in each move may not be properly evaluated. It also implies that MCTS as a whole may not be as suitable to Hanamikoji as initially perceived. The state space having a shallow and fixed depth but an exponentially increasing breadth contrasts the examples in Section 2, consisting of games with state spaces that are narrower, due to fewer options being available every turn, but also deeper, due to having a far higher turn limit, or lacking one altogether.

A final noteworthy observation is the smaller search trees in round 2, where these agents moved last. This is more than likely due to the opponent sometimes offering a Gift or Compete on turn 1 or 2. Search trees with a card offer in the root node are generally far narrower, due to fewer options being available for drawing than for making a move of your own.

Overall, move evaluations through MCTS appear to overall be worse than the regular Monte Carlo evaluations performed by the simple agent. In the next experiment, we will analyze an alternative method to perform searches of increased depth.

8.8 Recursive Rollouts

In the next experiment, we will be testing a modified simple agent that performs n_1 recursive rollouts, henceforth referred to as a recursive simple agent. These are rollouts that use two simple agents of n_2 regular rollouts. We will perform trials against a regular simple agent with 20 rollouts, adjusting n_2 between them.

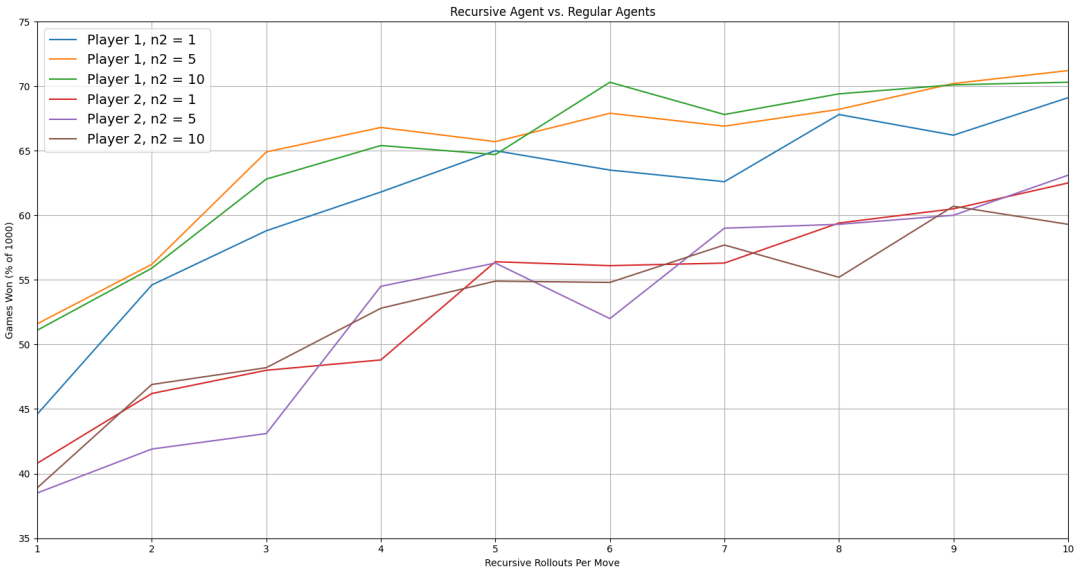

Figure 12: Recursive simple agents vs. simple agents (1000 games)

We see that n_1 remains the most significant parameter in the configuration of this agent, with both players having relatively similar results between various values of n_2. This could be due to the fact that n_2 only contributes to the evaluation of individual options in a turn, whereas n_1 defines the evaluation of said options in comparison to one another. In spite of this, the recursive simple agent does outperform the simple agent with very few recursive rollouts, implying that these rollouts are more informative than regular ones.

Another noteworthy observation is the persistent disparity in results between player 1 and 2. This further reinforces the notion that the simple agent's move evaluation algorithm is more suitable to player 1 than it is to player 2, as observed in Experiment 8.6, as well as the trend of more informative rollouts consistently favoring player 1. To test this further, we will test how they improve when cheating. n_2 will be set to 5 during these trials.

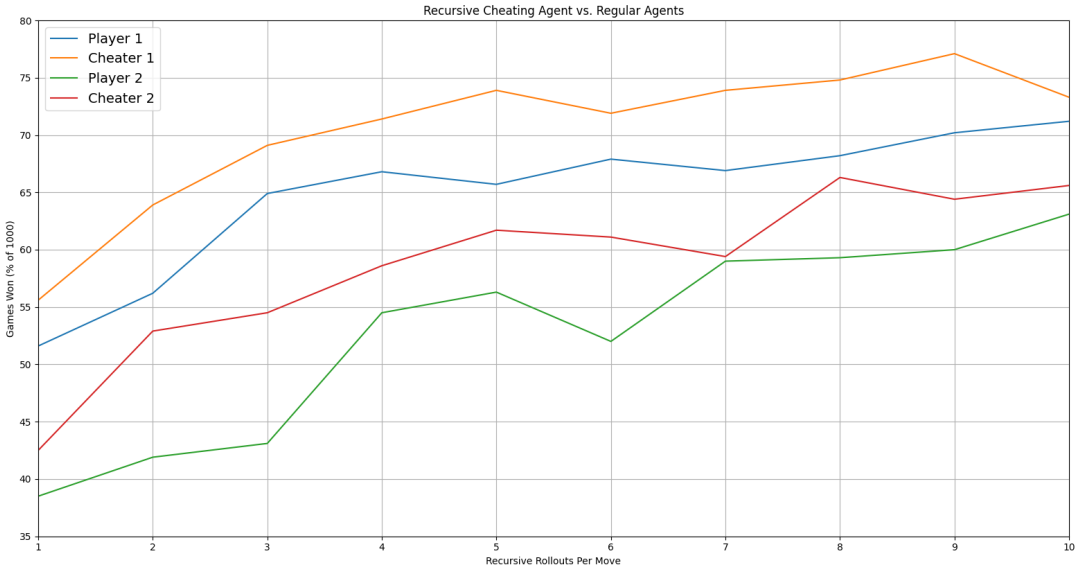

Figure 13: Cheating recursive simple agents vs. simple agents (1000 games)

The improved performance of cheaters aligns with the precedent set by Experiment 8.4, and shows that the most accurate agent of the ones designed in this research without any heuristics is the recursive simple agent. Despite this, it is not the most efficient agent. In the table below, All 4 agents are compared at their most optimal configurations in trials of 1000 games as player 1. All the agents are cheating and pre-sampling, as those appear to be universally beneficial to all agents.

agent	Parameters	W_1	$\mu(l \mid a = 1)$	Runtime (s)
random	none	6.9%	2.15	6.138
simple	$n = 50$	76.7%	2.07	11.481
MCTS	$n = 50, d = 2$	49.0%	1.97	22.139
Recursive	$n_1 = 9, n_2 = 5$	77.1%	2.02	808.963

Table 11: Performance statistics of each agent.

We can not observe any significant difference in performance between the simple and recursive simple agents. Conversely, we can see that the recursive simple agent demands roughly 70 times the runtime for a tenuous increase in accuracy. This implies that the simple agent is overall the most effective of the agents designed in this research, as searches of extended depth are prone to be either inaccurate, in the case of MCTS, or too slow, in the case of recursive rollouts. To conclude, we will observe the move order preferences of the recursive simple agent.

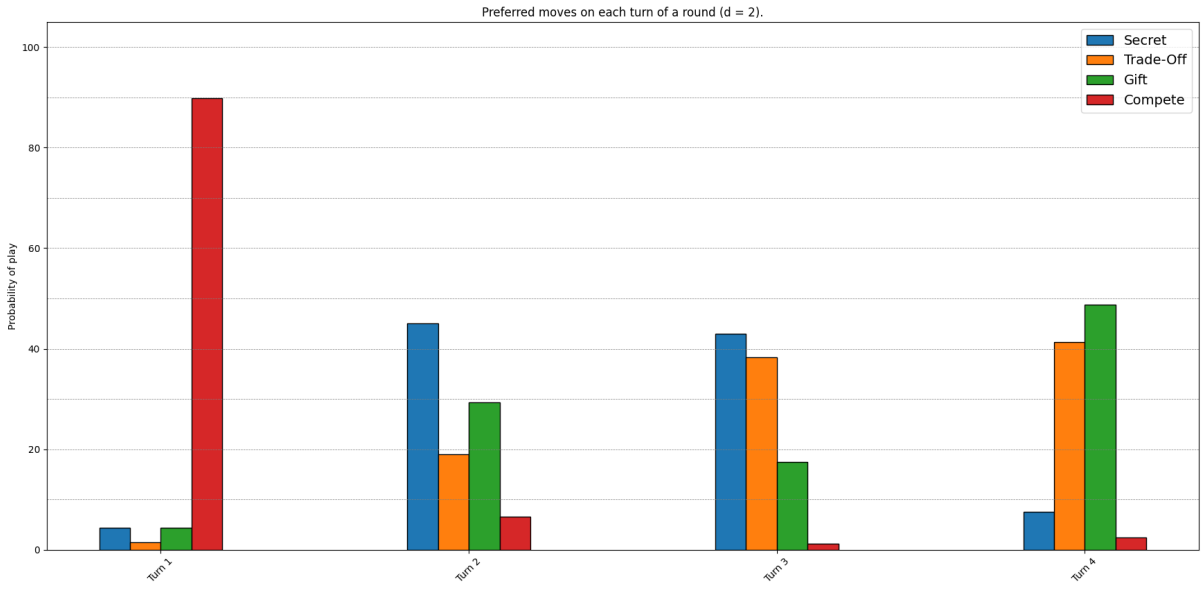

Figure 14: Most common moves on each turn by recursive simple agent.

Immediately we can observe a vastly different preference in move order from the recursive simple agent, almost exclusively choosing Compete on turn 1. This contradicts both the MCTS and simple agents, which would rarely choose Compete before turn 3. It also contrasts the results of Experiment 8.6, which showed that Compete was the least consistently successful first move. It also rarely plays Secret last and plays Trade-Off last far more often. This is consistent with the results of Experiment 8.6 which showed that Secret was the worst move to play last, and Trade-Off was the best. In general, its move choices in the last 3 turns of the round are also far more varied, in stark contrast to the other agents, which almost always played one of a very small set of move orders that were consistently favored in evaluation. This implies that there may not be a strictly optimal move order, but rather that there is a consistently "safe" move order for agents with simpler and less granular move evaluation algorithms.

9 Conclusion and Discussion

In this research, we implemented a modified game of Hanamikoji to be simulated by agents utilizing various resources and algorithms to optimize their decision-making. We determined that their performance far superseded that of an agent making moves at random and that their rollouts could be shortened by way of pre-sampling with little downside. We saw that their evaluations of moves, cards, and Victory Markers led to emergent patterns in play and performance that remained consistent throughout nearly all experiments. These patterns primarily emphasized a disadvantage for the player moving first in a given round, a general preference for even-valued Geishas over odd-valued Geishas, and a substantial advantage for a player who is cheating. We saw that MCTS is not as effective an algorithm as anticipated for the game, likely due to the breadth of its state space being more significant than its depth. Lastly, we saw that using recursive rollouts led to significantly different playing patterns, showing that there may not be one strictly ideal move order for all games.

The most straightforward avenue for further research is to remove the modifications made to the game in this research and observe how that may alter the results of the experiments performed. Additionally, the pre-sampling database was made by agents who were not pre-sampling themselves. Re-sampling the database, potentially numerous times, with different agents, could lead to the agents performing a rudimentary form of unsupervised learning, and the database itself changing noticeably. Agents with more advanced heuristics and optimization techniques could be made, potentially with unique logic for both players. The utility of MCTS could be further investigated, as the search depth remained limited in this research due to limited resources. Additionally, the MCTS algorithm could be tested in a top-down, pruning approach, rather than the bottom-up exhaustive search applied in this research, which was shown to be less effective than other methods. Lastly, the breadth and depth of the game's state space could be examined in more detail and potentially clarified with mathematical deduction that accounts for variance between turns.

Notation Glossary

Symbol	Meaning	Domain
t	Turn in a round	$\{1, 2, 3, 4\}$
r	Round in a game	$\{1, 2, 3, T\}$
l	Length of a game in rounds	$\{1, 2, 3, T\}$
$(l \mid \phi)$	Length of games with logical property ϕ	$\{1, 2, 3, T\}$
p	Player ID	$\{1, 2\}$
n	Rollouts used during a move evaluation	$\mathbb{N}$
d	Search depth of a move evaluation	$\mathbb{N}$
$G_{t,r}$	Game state at the end of turn t of round r	
a	Result of a regular game	$\{-1, 0, 1\}$
$a(G_{t,r})$	Result of a game played from a fixed state	$\{-1, 0, 1\}$
m_i	Victory Marker i	$i \in \{1, 2, \ldots, 7\}, m_i \in \{0, 1, 2\}$
W_p	General win rate of player p	$[0, 100]\%$
$W_p(r)$	Win rate of player p in round r	$[0, 100]\%$
$\mu(x)$	Mean value of aggregated data points x	$\mathbb{R}$

References

[Are22] Oleg Arenz. Monte carlo chess. B.S. thesis, Technische Universität Darmstadt, 2022.

[BPW+12] 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.

[CM07] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *arXiv preprint cs/0703062*, 2007.

[CZN11] Zhongjie Cai, Dapeng Zhang, and Bernhard Nebel. Playing tetris using bandit-based monte-carlo planning. In *In Proceedings of AISB 2011 Symposium: AI and Games (AISB 2011)*, 2011.

[DM] Vladimir Fedorovich Dem'yanov and Vasiliĭ Malozemov. *Introduction to minimax*.

[Fu18] Michael C Fu. Monte carlo tree search: A tutorial. In *2018 Winter Simulation Conference (WSC)*, pages 222–236. IEEE, 2018.

[GS11] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[Hmk17] Hanamikoji. BoardGameGeek, July 20 2017. BoardGameGeek ID 158600.

[KM75] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[KW09] Malvin H Kalos and Paula A Whitlock. *Monte carlo methods*. John Wiley & Sons, 2009.

[Nor19] Linus Norström. Comparison of artificial intelligence algorithms for pokémon battles. 2019.

[Skl99] David Sklansky. *The theory of poker*. Two plus two publishing LLC, 1999.

[VdBDR] Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-carlo tree search in poker using expected reward distributions. In *BNAIC 2009 Benelux Conference on Artificial Intelligence*, page 297. Citeseer.