Monte Carlo Search and Games Project:

Exploring Monte Carlo Search Variants for Indian Trick-Taking Card Game (Teen - Do - Paanch): From Perfect to Imperfect Information

Submitted By:

Kanupriya Jain

April 15, 2025

Contents

1 Introduction 2

2 Game Description 2

3 Game Implementation 3

4 Perfect Information Approaches 3
- 4.1 Win Rate Analysis 4
- 4.2 Time Efficiency 4

5 Imperfect Information Approaches 4
- 5.1 Win Rate Analysis 5
- 5.2 Time Efficiency Analysis 6

6 Evaluating Decision-Making Algorithms via Self-Play 6
- 6.1 Perfect Information SetUp 6
- 6.2 Imperfect Information with Basic Sampling (Redistribution) 6
- 6.3 Imperfect Information with Inference-Based Sampling 6
- 6.4 Results 7

7 Future Improvements 7

8 Conclusion 7

9 Appendix 8
- 9.1 Exploration Constant Vs Win Rate 8
- 9.2 Code Structure 9

References 10

1 Introduction

Trick-taking card games present a rich domain for artificial intelligence due to their mix of strategic depth, uncertainty, and evolving information states. Designing strong agents for such environments requires handling both perfect and imperfect information effectively.

In this work, we explore Monte Carlo-based decision-making techniques for the Indian trick-taking card game **Teen Do Paanch (TDP)**.

To tackle these challenges, we investigate a range of Monte Carlo Tree Search (MCTS) variants and adaptations. We begin by assuming perfect information, where all hands are known to the player, to evaluate baseline strategies using:

- Flat Monte Carlo simulations
- UCB-based Monte Carlo
- Upper Confidence bounds applied to Trees (UCT)
- Sequential Halving

We then extend our approach to the **imperfect information setting** using determinization techniques. First, we redistribute hidden cards randomly to simulate possible world states. We further enhance this by incorporating **inference-based redistribution**, where opponents' hands are sampled based on past observed actions, improving realism over naïve random sampling. Our goal is to understand how these methods perform under different information settings, analyze the tradeoffs involved, and evaluate their effectiveness in the unique context of Teen Do Paanch.

The code for all the implementation can be found in this github repository : Monte Carlo Search and Games

2 Game Description

Teen Do Paanch (Three-Two-Five) is a three-player trick-taking card game traditionally played using a custom 30-card subset of the standard 52-card deck. The deck includes all four suits, with cards ranging from 7 to Ace. However, the 7 of Clubs and 7 of Diamonds are excluded, reducing the total number of cards to 30.

In the traditional format, each player is assigned one of three roles per round:

- "Teen" (Three): must win at least 3 tricks,
- "Do" (Two): must win at least 2 tricks,
- "Paanch" (Five): must win at least 5 tricks.

These roles rotate across rounds to ensure fairness, so that each player assumes each role at least once. If a player fails to meet their goal, it implies another player has exceeded theirs, and that player gains an advantage in the next round. Notably, the "Paanch" player—having the most difficult task—traditionally selects the trump suit, which can beat cards from any other suit.

For simplicity, our implementation modifies several aspects of the original game:

- Roles and trick targets are not enforced.
- The trump suit and the starting player are chosen randomly at the beginning of each round.
- Players must follow suit if they can; otherwise, they may play any card.
- The winner of each trick leads the next, which dynamically updates the turn order during the round.

These simplifications preserve the strategic core of the game while making it more suitable for algorithmic experimentation.

3 Game Implementation

Game Representation and Simulation Environment To implement MCTS and evaluate different decision-making strategies, I first created a complete simulation environment for the card game Teen Do Paanch. The environment is modular and consists of three core components:

1. **`Card, Deck` Classes:** These represent the basic elements of the game. `Card` encapsulates suit, rank, and value, which is needed to compare cards during trick evaluation. `Deck` manages card creation, deck-specific constraints (e.g., removing some 7s), shuffling, and dealing cards to players. Code for this can be found in the notebook named **`Card_and_Deck.py`**.

2. **`Player, Game` Classes:** These simulate the game logic. `Player` maintains a player's hand, role (Teen/Do/-Paanch), tricks won, and implements legal move generation based on current trick state and trump suit. `Game` holds the full game state including the players, trump suit, cards on the table, turn logic, and played cards. This class initializes the full game setup. The code for these classes can be found in **`Game_and_Player.py`**.

3. **`GameState` Class:** This class is designed to represent state of the game. It stores Stores an abstracted version of the full game state (player hands, *current_player_index*, table cards, etc.) It has the function `apply_move` that changes cards in the hands of player, cards on table. It also has the function `is_terminal` which decides if the round has ended or not. It also gives what legal moves are available to current player through the function `get_legal_moves()`. Supports state redistribution, both: Randomized (basic Perfect Information Monte Carlo) Inference-based (using trick history to constrain hidden information). The redistribution is important for dealing with imperfect information in order to sample determinized state. The code for this class is present in **`GameState.py`**.

Further code structure can be found here in the section 9.2.

4 Perfect Information Approaches

Now, we have defined our games. We will explore the perfect information approaches i.e. the scenario where each player knows everything including the cards in opponents hands. We implemented four approaches for dealing with perfect information-

1. Flat Monte Carlo
2. UCB
3. UCT
4. Sequential Halving

For each method (e.g., Flat MC, UCT, etc.), run it in two matchups. First, the player using the algorithm competes against two players who play randomly from their legal moves. For the second case, the player using the algorithm compete against players who always play the highest-ranked legal card. In each case, we randomly choose the player. Our purpose is to measure how well our method does in a simple baseline environment as well as against a greedy and stronger heuristic baseline. We played 500 games in each case and kept number of simulations to be 100 in each case. We used win rate as our metric of evaluation. For UCT, we explored two reward strategies. In the first one, we tried to do max^n approach where after each simulation, the reward for each player is computed based on the number of tricks they won, normalized by the maximum possible number of tricks (10). This version is cooperative-agnostic and non-zero-sum. Each player tries to maximize their own trick count, regardless of others.In the second strategy we tried to the minimax approach where, we use a zero-sum reward structure where one player's gain is others' loss. The current player's reward is positive and all others are penalized with the negative of that same reward.

For UCB and UCT,We tested various values of exploration constants for 100 games and then used that value to run the algorithms for 500 games against random and rule based play. We found that for UCB the best value was $c = 0.7$ which is very close to the suggested value $\frac{1}{\sqrt{2}} \approx 0.707$. For UCT, for the first approach (minimax) we used $c = 1.2$ and for the second approach (max^n) we used $c = 0.9$.

Opponent Type	**Flat Monte Carlo**	**UCB**	**UCT (minimax)**	**UCT(max^n)**	**Sequential Halving**
Random Players	59.20%	55.80%	55.60%	58.40%	**59.40 %**
Rule-Based Players	57.2%	58.40%	56.00%	**62.60%**	54.60%

Table 1: Win Rate for different methods for Random and Rule Based Play

Opponent Type	**Flat Monte Carlo**	**UCB**	**UCT (minimax)**	**UCT(max^n)**	**Sequential Halving**
Random Players	1197.876 s	301.06 s	320.49 s	290.73 s	256.84 s
Rule-Based Players	1176.456 s	305.83 s	283.52 s	298.33 s	253.17 s

Table 2: Time Taken by each method to run 500 games

We evaluated five Monte Carlo-based algorithms against two types of opponents — Random and Rule-Based — over 500 games each. The results, summarized in Tables 1 and 2, reveal important insights into the performance and efficiency of each method.

4.1 Win Rate Analysis

Against Random players, all methods performed significantly better than random play (33.3%), with Flat Monte Carlo and Sequential Halving achieving the highest win rates at 59.2% and 59.4%, respectively. This indicates that even without opponent modeling, extensive simulation can yield effective strategies. Against Rule-Based players, the UCT (max^n) algorithm outperformed all others with a 62.6% win rate, suggesting that it benefits from modeling all players' decisions independently. Interestingly, UCT (minimax) and UCB achieved similar or slightly lower performance, which may be due to their assumptions of strictly adversarial opponents (minimax) or reliance on a fixed exploration strategy (UCB), which may not adapt well to non-random structured play.

One thing to note here is the fact that while we expected Monte Carlo-based agents to perform worse against rule-based opponents due to their more strategic behavior, an unexpected trend emerged: all tree-based methods and exploratory methods like UCB and UCT (max^n) actually performed better against rule-based players than against random ones. This may be due to the fact that random opponents introduce more uncertainty into the simulation, which can destabilize the value estimates in MCTS or UCB, especially when only 100 simulations are used per move. Rule-based players, while better than random ones, follow predictable patterns. This can sometimes be easier to exploit for search-based methods like UCB and UCT, which may learn to anticipate and counter these deterministic behaviors, especially over repeated simulations. Unlike UCB and UCT, Flat Monte Carlo and Sequential Halving do not improve against rule-based opponents. This is expected because these methods lack mechanisms to model opponent behavior or simulate deep, adaptive sequences.

4.2 Time Efficiency

Flat Monte Carlo was the most computationally expensive, taking roughly 4 times longer than tree-based approaches due to its lack of tree reuse. In contrast, Sequential Halving was the fastest method across both opponent types, while still maintaining competitive win rates, especially against Random players. The UCT-based methods (UCB, minimax, max^n) offered a good trade-off between performance and efficiency.

5 Imperfect Information Approaches

Up to this point, all methods assumed full observability—i.e., that players have access to the complete game state, including the cards held by opponents. However, in trick-taking games like Teen Do Paanch, players operate under imperfect information, as the opponent's hands are hidden. To handle this, we implemented a sampled determinization approach. Here, the agent generates multiple plausible versions of the game state by redistributing hidden cards among opponents, and evaluates actions under these sampled "possible worlds." We explored two methods for sampling:

1. **Basic Sampling (`redistribute`):** Hidden cards of opponents are randomly shuffled and redistributed to the opponents, without considering prior trick history.

2. **Inference-Based Sampling (`redistribute_with_inference`)**: We improve realism by inferring suit constraints from trick history. For instance, if a player fails to follow suit when they could have, we infer they no longer possess cards of that suit, and avoid assigning such cards during redistribution.

Now, we extended UCB, UCT and Sequential Halving to imperfect information, we followed the determinized sampling approach. For each decision, we sampled multiple plausible world states consistent with the current information (using `redistribute` or `redistribute_with_inference`). Each determinization was treated as a perfect information instance, and the algorithm (UCB or Sequential Halving or UCT) was run independently. The final move was selected via majority vote over the best moves across all samples.

In this case, we only studied 100 games of each algorithm with `max_steps=100` (number of simulations needed for each algorithm) and `num_samples=20` (i.e. number of determinized sampled will be 20) due to limited computational resources. For UCT, we kept our reward strategy where each player tries to maximize their own rewards i.e. max^n approach, as this variant consistently yielded better performance in our earlier experiments. The same exploration constants used in the perfect information case were retained for consistency—specifically, we used $c = 0.9$ for UCT and $c = 0.7$ for UCB.

We excluded Flat Monte Carlo from the imperfect information experiments due to its significantly higher computational cost. In earlier evaluations under perfect information, Flat Monte Carlo was observed to take approximately four times longer than the other approaches. Since the determinization strategy already introduces additional overhead (due to sampling and repeated simulations), we limited our study to UCT, UCB, and Sequential Halving, which offered a more practical balance between performance and computational feasibility.

Results against Random Play

Sampling Type	**UCB**	**UCT(max^n)**	**Sequential Halving**
Basic Sampling	56.00 %	50.00%	**59.00 %**
Inference Based Sampling	58.00 %	58.00%	**61.00%**

Table 3: Win Rate for different methods against Random Play using Basic and Inference Based Sampling

Sampling Type	**UCB**	**UCT(max^n)**	**Sequential Halving**
Basic Sampling	1183.36 s	1170.43 s	942.78 s
Inference Based Sampling	1208.33 s	1181.52 s	1003.66 s

Table 4: Time Taken for different methods against Random Play using Basic and Inference Based Sampling

Results against Rule Based Play

Sampling Type	**UCB**	**UCT(max^n)**	**Sequential Halving**
Basic Sampling	**57.00 %**	48.00 %	51.00 %
Inference Based Sampling	**57.00%**	55.00%	**57.00%**

Table 5: Win Rate for different methods against Rule Based Play using Basic and Inference Based Sampling

Sampling Type	**UCB**	**UCT(max^n)**	**Sequential Halving**
Basic Sampling	1207.55 s	1164.62 s	947.82 s
Inference Based Sampling	1187.59 s	1174.24 s	977.82 s

Table 6: Time Taken for different methods against Rule Based Play using Basic and Inference Based Sampling

5.1 Win Rate Analysis

- Across both Random and Rule-Based opponents, we observe that Inference-based sampling consistently improves performance over basic sampling. For example, UCT(max^n) sees a 8% increase in win rate vs. Rule-

Based players (from 48% to 55%).Sequential Halving shows improved robustness, increasing win rate from 51% to 57% in Rule-Based settings. Even UCB benefits slightly, maintaining or improving its win rate across both opponent types.

- Sequential Halving consistently performs best or near-best, especially when inference is used. We observe that against random play Sequential Halving performs best and it has highest win rate against Rule Based Play using Inference Based Sampling.
- Interestingly, in the case of Rule-Based opponents, UCB achieves the highest win rate (57%) among the three methods, even slightly outperforming Sequential Halving with inference. We again observe that win rate for UCB and UCT increased against rule based play (as happened before in the case of perfect information), while it decreased for Sequential Halving. This could be attributed to their exploration strategies, which allow them to better adapt to non-random, structured opponent behavior. On the other hand, Sequential Halving, which focuses more aggressively on eliminating weaker moves early, may be more susceptible to errors in the early stages of evaluation. As a result, its performance slightly declines against stronger opponents, despite improving under inference-based sampling.

5.2 Time Efficiency Analysis

From the tables above, we observe that Sequential Halving is consistently the most time-efficient method across both Basic and Inference-Based Sampling, regardless of opponent type. This aligns with the nature of the algorithm, which strategically narrows down the set of candidate actions early by allocating simulations more efficiently rather than exploring all options equally. We observed the same in the case of Perfect Information. We also observe that Inference-Based Sampling slightly increases the runtime for all methods due to the added logic of suit-based constraints during card redistribution, though the increase is very small.

6 Evaluating Decision-Making Algorithms via Self-Play

We conducted three key experiments to evaluate different decision-making algorithms that we studied under both perfect and imperfect information settings. Till now, we evaluated the algorithms against random and rule based play. Now, in order to have a better idea on the performance, we let the algorithms play against each other in a self-play environment. This setup provides a more robust and meaningful analysis.

6.1 Perfect Information SetUp

In the first experiment, we assumed that all players had full access to the game state, including the cards in the hands of all opponents. This unrealistic but informative scenario allowed us to benchmark the algorithms under ideal conditions. We compared the following methods: **UCB**, **UCT** (max^n variant) – Each player attempts to maximize their own reward and **Sequential Halving**. We played 100 games with 100 number of simulation for each move.We compared three strategies for a three player game. The players were randomly assigned the strategy each time before the start of the game in order to make sure that there is no bias.

6.2 Imperfect Information with Basic Sampling (Redistribution)

In the second experiment, we modeled a more realistic game scenario where players do not know the opponents' cards.To address hidden information, we used a determinization approach—sampling possible world states consistent with known information by randomly redistributing hidden cards (via the `redistribute` method). No inference from previous tricks was used in this baseline version. We compared **UCB**, **UCT** (max^n variant) and **Sequential Halving** versions of determinizations by applying them to sampled states as we did before in Section 5.

We played 100 games with `num_samples=20` and `max_steps` = 100 for each move. As before, to compared three strategies for a three player game we randomly assigned the strategy to the players each time before the start of the game. We used the basic sampling strategy for determinization state which ignored prior trick history and simply reshuffled hidden cards uniformly at random. This gave us a baseline for imperfect information performance.

6.3 Imperfect Information with Inference-Based Sampling

In the third experiment, we extended the second approach by incorporating simple inference mechanisms (via `redistribute_with_inference`) to improve the realism of determinized worlds.

6.4 Results

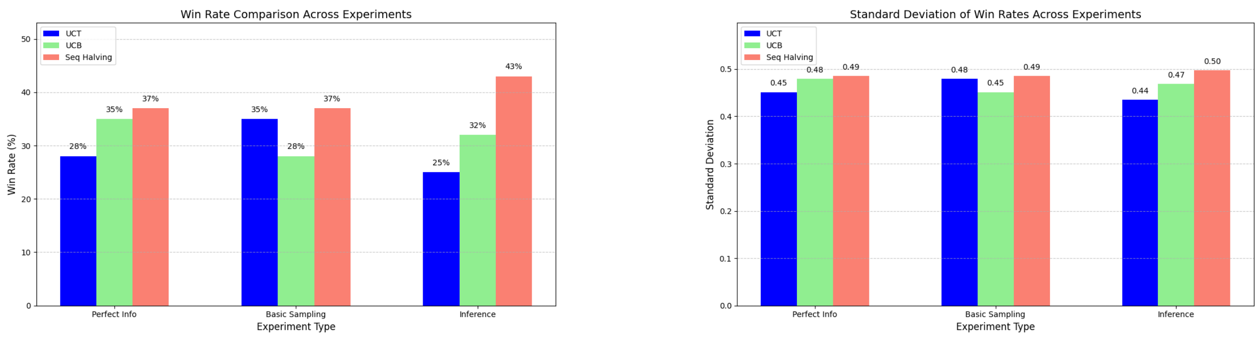

Figure 1: Results for Self Play

Observations: Across all experimental setups, Sequential Halving consistently demonstrated strong performance, especially in imperfect information scenarios where inference was used. Its combination of win rate and time efficiency makes it the most practical choice among the three. While UCB showed early strength in perfect information, it did not generalize as well under uncertainty. UCT ($\max^n$) performed reasonably but often lagged behind in win rate.

In the perfect information setting, UCB (35%) and Sequential Halving (37%) perform similarly. In the basic sampling setup, UCT (35%) and Sequential Halving (37%) are also close. However, with inference-based sampling, Sequential Halving clearly outperforms the others with 43%, while UCB and UCT drop to 32% and 25%.

Interestingly, UCB and UCT performed well against rule-based (strategic) opponents in earlier experiments, which may explain their competitive performance here where all agents use strong strategies. However, when Sequential Halving was given the inference, it was able to exploit much better.

7 Future Improvements

We used Determinization to handle imperfect Information but it suffers from **Strategy fusion** i.e. an AI agent cannot make different decisions from different states in the same information set; however, the deterministic solvers can and do make different decisions in different determinizations and **Non-locality** i.e. some determinizations may be vanishingly unlikely (rendering their solutions irrelevant to the overall decision process) due to the other players' abilities to direct play away from the corresponding states. Some of the further works to improve our current implementation include -

- Information Set UCT (ISUCT)
- Monte Carlo Tree Search with Alpha-Beta Pruning
- Deep Reinforcement Learning (DRL)
- Bayesian Inference and Opponent Modeling

8 Conclusion

We evaluated Flat Monte Carlo, UCB, UCT (minimax and max^n), and Sequential Halving across perfect and imperfect information settings, against both random and rule-based opponents, and in self-play.

Against random opponents, all methods performed well, with Flat Monte Carlo and Sequential Halving showing slightly higher win rates. Against rule-based play, UCT (max^n) stood out with 62.6%, and both UCB and UCT improved compared to their random-play performance, suggesting their strength in strategic settings.

In imperfect information, using inference-based sampling generally improved win rates over basic sampling—especially for Sequential Halving, which reached 57% against rule-based play. However, this came with increased computation time for most methods. In the self-play experiment, Sequential Halving seems to outperform all the other algorithms we tested for our game of Teen-Do-Paanch.

9 Appendix

9.1 Exploration Constant Vs Win Rate

In order to find the best value of exploration constant, we tested different values of $c\_values = [0.1, 0.3, 0.5, 0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 1.7, 2.0]$ for UCB and both approaches of UCT explained above. We plotted exploration constant against win rate and time it took to run 100 games for each value of c.

For UCB

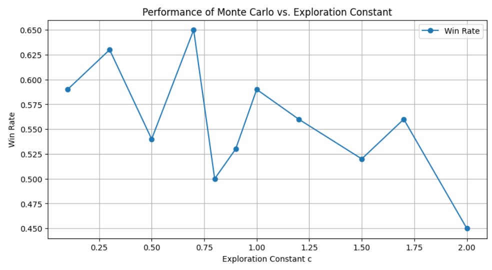

Figure 2: Exploration Constant Vs Win Rate

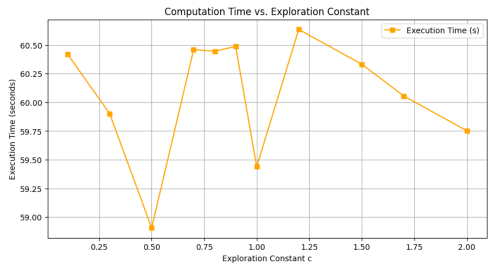

Figure 3: Exploration Constant Vs Time Taken to run 100 games

Figure 4: Results for UCB approach

UCT (max^n)

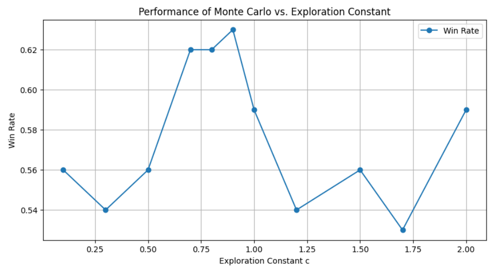

Figure 5: Exploration Constant Vs Win Rate

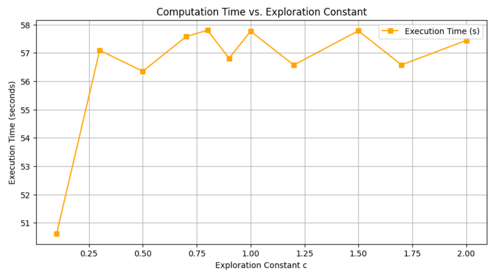

Figure 6: Exploration Constant Vs Time Taken to run 100 games

Figure 7: Results for UCT(max^n) approach

UCT (minimax)

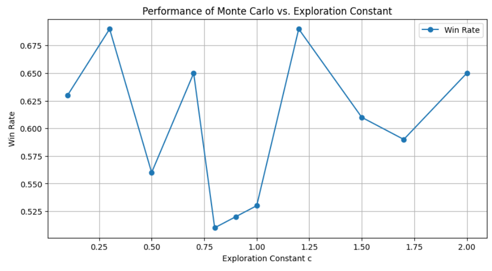

Figure 8: Exploration Constant Vs Win Rate

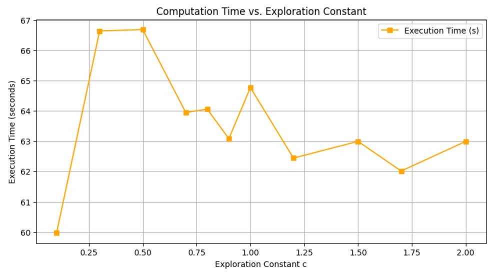

Figure 9: Exploration Constant Vs Time Taken to run 100 games

Figure 10: Results for UCT(minimax) approach

9.2 Code Structure

1. There are three main files: `Card_and_Deck.py`, `Player_and_Game.py`, and `GameState.py`, which contain the core classes described in Section 3.

2. For UCT-based methods, we implemented an `MCTSNode` class that represents a single node in the Monte Carlo Tree Search (MCTS) tree. Each node encapsulates a game state and tracks relevant information such as visit counts, accumulated rewards, and child nodes. It also includes methods for expansion, simulation, backpropagation, and selection (tree policy). This class is reused across all notebooks that involve UCT for comparison.

3. `Final_Flat_Monte_Carlo.ipynb` contains the Flat Monte Carlo implementation. `UCB.ipynb` includes the UCB method. `UCT_Final_maxn.ipynb` and `UCT_Final_minimax.ipynb` implement UCT with max^n and minimax evaluation, respectively. `Sequential_Halving.ipynb` contains the Sequential Halving implementation. Each of these notebooks also includes evaluation results against Random and Rule-Based opponents, corresponding to Tables 1 and 2.

4. Determinized versions of each algorithm using basic sampling are implemented in: `Determinized_UCB.ipynb` (for UCB), `Determinized_UCT_maxn.ipynb` (for UCT max^n), and `Determinized_Sequential_Halving.ipynb` (for Sequential Halving). Each notebook contains the method's implementation and its performance against Random and Rule-Based opponents.

5. Similarly, inference-based determinization is implemented in: `Inference_based_Determinized_UCB.ipynb`, `Inference_based_Determinized_UCT.ipynb`, and `Inference_based_Sequential_Halving.ipynb`. Each notebook includes both the algorithm and its results against Random and Rule-Based opponents.

6. Results for Tables 3, 4, 5, and 6 are drawn from the notebooks listed in points 4 and 5.

7. Self-play results, where different algorithms play against each other, are found in:

   - `Perfect_Info_Self_Play_Comparison.ipynb`: all players have full knowledge of all cards.
   - `Imperfect_Info_(no_inference)_SelfPlay.ipynb`: players do not see opponents' cards, and determinization does not use trick history.
   - `Imperfect_Info_(with_inference)_SelfPlay.ipynb`: players do not see opponents' cards, but inference-based determinization is used.

   These notebooks include results for Figure 1.

References

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

Cazenave, T. (n.d.), ‘Lecture notes on monte carlo search methods’. Lecture Notes (M2 IASD), Université Paris-Dauphine, PSL.
URL: *https://www.lamsade.dauphine.fr/ cazenave/MonteCarlo.pdf*

Whitehouse, D., Powley, E. J. and Cowling, P. I. (2011), ‘Determinization and information set monte carlo tree search for the card game dou di zhu’, *2011 IEEE Conference on Computational Intelligence and Games (CIG’11)* pp. 87–94.
URL: *https://api.semanticscholar.org/CorpusID:15054631*