# Experimental Analysis of AI-powered Algorithms for Optimal Mendikot Gameplay

Anway Shirgaonkar
*College of Engineering*
*Northeastern University*
Boston, MA
shirgaonkar.a@northeastern.edu

Neeraj Sahasrabudhe
*College of Engineering*
*Northeastern University*
Boston, MA
sahasrabudhe.n@northeastern.edu

***Abstract*—Mendikot is a popular multi-player strategic team-based trick-taking card game originating in South Asia. The objective of the game is to win the most number of tricks with a 10 in it. Previous work on Mendikot does not exist, therefore developing an AI agent for it means starting from scratch in terms of environment and algorithm design. The partially observable nature of the game at each timestep complicates this task and adds another level of complexity to the decision-making process. The task is made more challenging by the huge state space and dynamic action space that depend on the agent's present state. We have attempted to solve this problem using two approaches; Monte Carlo Tree Search (MCTS) and Function Approximation. Applying these algorithms on a novel task was a big learning curve for us**

***Index Terms*—Artificial Intelligence, Card Game, Function Approximation, Monte Carlo Tree Search (MCTS)**

## I. Introduction

Reinforcement learning is a major area of machine learning. Reinforcement learning agents learn by exploring their environments on their own. The environment gives rewards for actions of the agents and the concern of the agents is to maximize the total rewards. Game environments (including card and board games) are appropriate for reinforcement learning. They respond to the actions of reinforcement learning agents by checking if the agents satisfy the objectives of the game or determining how close they are. [2] There are several studies in literature in which reinforcement learning techniques are used to develop game playing agents. Arthur Samuel's checkers playing program [3], [4] is one of the earliest work that uses reinforcement learning. His work forms the basis of reinforcement learning. TD-Gammon [5], [6] is one of the most successful applications of reinforcement learning for games. It is a world-class backgammon playing program developed using temporal difference learning techniques.

Furthermore, advances in AI research have developed strong adversarial tree search methods, specifically the Monte Carlo Tree Search (MCTS) family of algorithms. These algorithms have proven to be very efficient in combinatorial games such as the board game Go, where previous attempts at artificial intelligence did not produce results capable of challenging human players. Since the official proposal of MCTS in 2006 many enhancements and variations have been studied and experimented on several game domains, including games of hidden information, where tree search complexity is relatively high and classic algorithms, such as expectimax, can not give the optimal move in a feasible amount of time, due to the large branching factor of hidden information game trees. [7] By analysing trick-taking card games such as Sueca and Bisca, where players take turns placing cards face up in the table, there are similarities which allow the development of a MCTS based implementation that features enhancements effective in multiple game variations, since they are non deterministic imperfect information problems. Good results have been achieved in this domain with the algorithm, in games such as Spades [8] and Skat [9].

In Mendikot, teams of two compete, accommodating any even number of players. A standard deck of 52 playing cards are randomly distributed among players, who then take turns playing cards to form tricks. The primary aim is for a team to win the most tricks containing ”tens.” In the event of a tie in tens, the winner is determined by the number of tricks won overall. Our work aims to evaluate the performance of the two algorithms RL based linear function approximation and MCTS. We started with formalizing the game into mathematical model which was required to develop the environment for this game. We have tried to ”vectorize” the entire game in a 52x15 matrix, where the rows are each of the 52 playing cards, and the columns are unique features associated with that card. We will discuss this in more detail further. This matrix encodes all the essential information required to know the current state of the game at any point in time.

## II. Game Description

Mendicot also recognized as Mendikot or Mindi, stands out as a popular card game played across diverse regions, particularly in India and Nepal. It is believed that it has its origins in the Indian states of Maharashtra and Gujarat. The game shares some similarities with 'Dehla Pakad' which is fairly popular in these regions.

The game of Mendikot is played with a standard deck of 52 cards between 2-4 players. While sitting in a circle facing inwards, players sitting opposite to each other form a team. One of the unique characteristics of the game is that it is a team game. Every player's move aligns with a strategy to maximize the team's reward and not their individual reward. This game intricately weaves together strategy and foresight, offering a riveting experience that captivates all participants.

Mendikot is a trick-taking card game where every team tries to win the game by implementing a thoughtful strategy. The total cards in the deck are distributed among the participating players equally. A trick is defined as a unit game-play which has a a set of cards to which each player contributes by playing 1 card. So, for a game played between 4 players, 1 trick would consist of 4 cards. The winner of the previous trick gets an opportunity to play the first card of the next trick. The suit of the first card is considered as the 'trick suit'. All the other players who play after the first player have to follow the 'trick suit'. After every player plays their card, the card with the maximum value (where 2 being the lowest and ace being the highest) wins the trick.

Moreover, the game also has a Trump suit. The Trump suit can either be pre-defined before starting a game or can be decided in-game. In our implementation, the Trump suit is decided randomly before a game begins. Alternatively, some variations of the game begin with no Trump suit chosen initially. When a player is not able to follow a suit that was played first in the table, the suit of the card they choose to play becomes the trump suit for that round. The trump suit card overpowers any other card in the game. Thus, the Trump suit plays a key role in deciding the optimal strategy while playing Mendikot.

The crux of the game lies in accumulating the maximum possible tricks with 10. The players generally try to collect the highest number of tens from all four suits by either supporting their teammates or by maximizing their individual wins. The game continues until all the cards in hand are played by every player. Ultimately the team with the highest number of tricks with 10 wins the game.

## III. Environment

### A. State space, Action space and Rewards

The state of a `PLAYER` is a Nx9 matrix, where N is the number of cards that are in play, i.e. given by the `CARD_FOR_PLAYING` feature, and the 9 columns are the features `CARD_CURR_TRICK_PLAYER`, `CARD_PREV_TRICK_PLAYER`, (4 each for each of the players) and `CARD_TRUMP`. the size of the state space is enormous and given by the formula in 1. The rewards given to the agent are:

- +5 : Trick won with a 10 (By agent or teammate)
- +1 : Trick won without a 10 (By agent or teammate)
- -5 : Trick won with a 10 (By opponents)
- -1 : Trick won without a 10 (By opponents)

$$\underbrace{\boxed{C^{4}_{1}}}_{\text{Trump suite}} * \underbrace{\boxed{4!\,C^{16}_{4}}}_{\text{Current trick}} * \underbrace{\boxed{[C^{12}_{0} + 4!\,C^{12}_{4} + 8!\,C^{12}_{8} + 12!\,C^{12}_{12}]}}_{\text{Previous trick}}$$

Fig. 1. Size of the state space

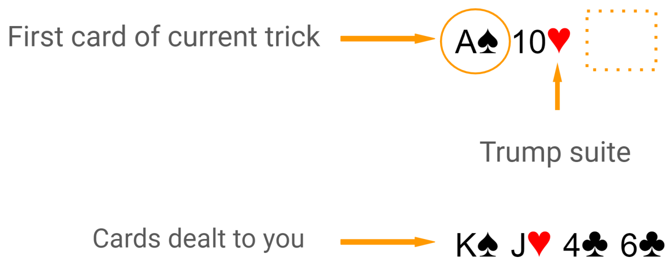


Fig. 2. Dynamic action space

The action space is a bit more complicated. The action refers to a card that a player can play in the current trick. The eligible cards that can be played in a trick are dictated by the rules of the game, and depend on as stated in the rules below. You can see an example of this in 2

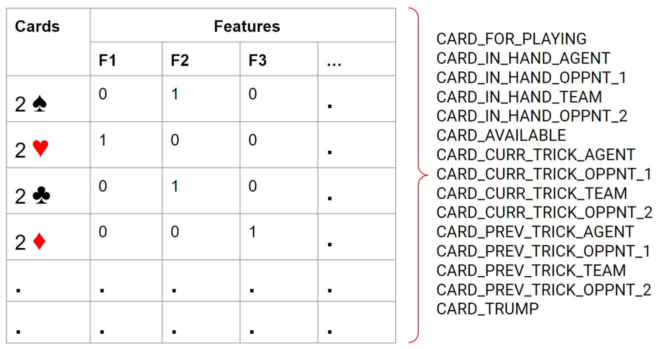
- The first person to play in a trick can play any one of the cards that are dealt to them. There is no restriction here, but the next players need to follow some rules. The suite of the first card played in a trick is called the trick's suite.
- As long as a player has cards belonging to the trick suite, they can only play any of those cards.
- If a player does not have a card belonging to the trick suite, then they can play a card from the trump suite.
- If a player has does not have any card belonging to the trick suite or the trump suite, then they can play any of the cards dealt to them. In this case, they cannot win the trick at all.

### B. Formal Definition and Representation

As discussed earlier, we started with developing the environment for Mendikot. We have made some assumptions in our version of Mendikot to simplify the development namely:

- Only four players play the game `AGENT`, `OPPONENT_1`, `TEAMMATE`, `OPPONENT_2`
- The trump suite is selected randomly at the start of a trick
- The agent is the player being trained. Other players follow a random policy
- The agent has the ability to remember all cards played in the current and previous tricks (which is a rare human ability)

The game is represented by a 52x15 matrix. Each of the 52 rows correspond to the 52 playing cards and the 15 columns represent the boolean features associated with that card as shown in 3. Below is the description of all the features that



| Cards | Features | | | |
|---|---|---|---|---|
| | F1 | F2 | F3 | ... |
| 2 ♠ | 0 | 1 | 0 | . |
| 2 ♥ | 1 | 0 | 0 | . |
| 2 ♣ | 0 | 1 | 0 | . |
| 2 ♦ | 0 | 0 | 1 | . |
| . | . | . | . | . |
| . | . | . | . | . |

CARD_FOR_PLAYING
CARD_IN_HAND_AGENT
CARD_IN_HAND_OPPNT_1
CARD_IN_HAND_TEAM
CARD_IN_HAND_OPPNT_2
CARD_AVAILABLE
CARD_CURR_TRICK_AGENT
CARD_CURR_TRICK_OPPNT_1
CARD_CURR_TRICK_TEAM
CARD_CURR_TRICK_OPPNT_2
CARD_PREV_TRICK_AGENT
CARD_PREV_TRICK_OPPNT_1
CARD_PREV_TRICK_TEAM
CARD_PREV_TRICK_OPPNT_2
CARD_TRUMP

Fig. 3. Representation of Mendikot as a matrix

are associated with a card.
NOTE: Wherever you see a feature ending with the word `PLAYER`, assume that there are actually 4 features for each player `AGENT`, `OPPONENT_1`, `TEAMMATE`, `OPPONENT_2` respectively.

- `CARD_FOR_PLAYING`: True if the card is considered for playing in the game, else False. The game can be played in a minimum of 16 cards (all suits of `10`, `Ace`, `King`, `Queen`) and a maximum of 52 cards (complete deck).
- `CARD_IN_HAND_PLAYER`: True if the card is dealt to the player, else False.
- `CARD_AVAILABLE`: True if the card is valid for playing in the current trick, else False.
- `CARD_CURR_TRICK_PLAYER`: True if the card is played by the `PLAYER` in the current trick, else False.
- `CARD_PREV_TRICK_PLAYER`: True if the card is played by the `PLAYER` in any of previous tricks, else False.
- `CARD_TRUMP`: True if the card belongs to the trump suite, else False.

As far as our Mendikot class is concerned, we have developed a Gymnasium-style class. Gymnasium is a maintained fork of OpenAI's Gym library. The Gymnasium interface is simple, pythonic, and capable of representing general RL problems [10]. Hence, in accordance to the Gymnasium style wrapper, we included the primary functions `step()` and `reset()` in the Mendikot class.

- `reset()`: This function must be called before the start of any game. The primary objective of this function is to randomly shuffle the cards and distribute it equally among the players as per number of cards in play. Secondly, this function randomly selects a trump suite for the first trick. In a nutshell, this function would update the `CARD_IN_HAND_PLAYER`, `CARD_FOR_PLAYING` and the `CARD_TRUMP` features in the game matrix.
- `step()`: Every player needs to call this function in order to play a card in a trick. This function has the ability to update the game matrix as per the complex rules of the game, and upon completion of the trick, it also evaluates the cards in that trick and returns the winner of the trick.

In conjunction to this, there are several helper functions which help update the state of the game

- `update_trump`: Update trump suite
- `reset_trick`: Update game matrix to reset trick.
- `get_render_str`: Get string to render on the CLI.
- `get_card`: Get card and suite corresponding to a row in the game matrix.
- `get_cards_in_play`: Get cards which are considered for playing in the current game.
- `get_cards_in_trick`: Get cards played in the current trick.
- `get_cards_in_hand`: Get the cards dealt to a player
- `get_cards_in_prev_trick`: Get cards played by players in the previous trick.
- `get_trump_in_trick`: Get the trump cards played in the current trick.
- `get_winner`: Get the winner or the current trick.
- `get_state`: Get the state of the player.
- `get_available_cards`: Get the cards which can be played in the current trick.
- `get_game_winner`: Get the winner of the game.
- `get_reward`: Get the reward of the current trick
- `is_trick_empty`: Returns True if no card is played in the current trick, else False
- `is_ten_in_trick`: Returns True if 10 is played in the current trick, else False
- `is_trick_complete`: Returns True if all players have played cards in the current trick, else False.
- `is_card_available`: Returns True if a card can be played in the current trick, else False.
- `is_game_complete`: Returns True if the game is complete, else False.
- `update_score_and_reward`: updates the score matrix with the rewards for the corresponding players.
- `evaluate_trick`: Evaluate the current status of the trick and determine the winner on completion of the trick.

### C. Environment Testing

After developing the environment, it was crucial to test the environment to ensure that the intricate details and rules of the game were being enforced. In order to achieve this, we decided to implement three strategies as listed below:

*1) Manual Agent:* We developed a Command Line Interface (CLI) which allowed us to test the game-play by actually playing the game. This proved helpful while determining minor bugs in the environment and also while checking if the agent is playing logically.

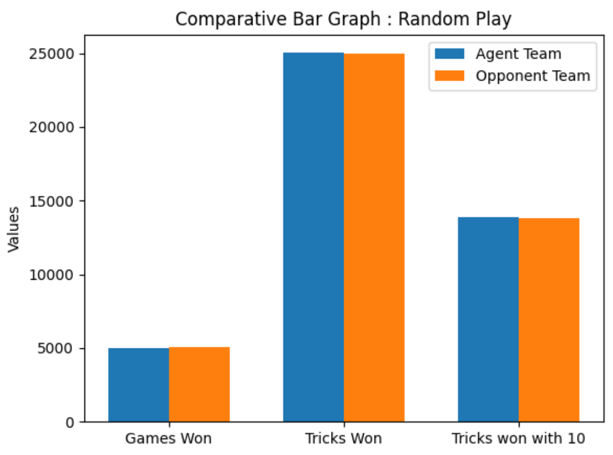


Fig. 4. Comparative analysis for Random Agent

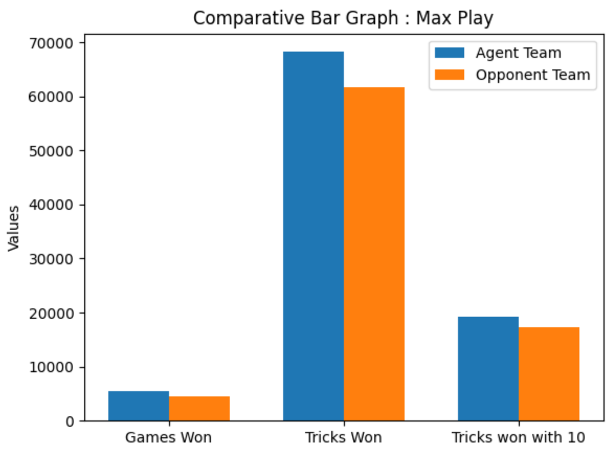


Fig. 5. Comparative analysis for Maximum Strategy Agent

*2) Random Agent:* In this approach, we used random action for each player playing the game. This allowed us to infer the correctness of the developed environment statistically. We ran the random agent for 10,000 games to infer that the number of wins for each team was approximately equal. This statistical evidence helped us prove that our environment is unbiased and fair. The results can be seen in Figure 4

*3) Maximizing Agent:* In this approach, we used a maximizing policy for the agent and a random strategy for all the other players in the game. We again ran the agent for 10,000 games. These results can be seen in Figure 5. It can be seen from the results that the agent's team won more games than the opponent's team. This was observed because the maximizing policy helps the agent in winning more tricks as opposed to a random policy enforced by the opponent's team.

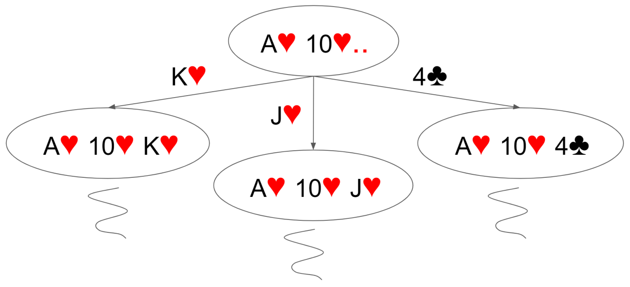


Fig. 6. Building a tree for MCTS

## IV. Algorithms

### A. Monte Carlo Tree Search

Now we understand the fact that we need something smart to navigate through a gigantic state space of Mendikot1. MCTS is one such smart algorithm that gives us a way out.Essentially, MCTS uses Monte Carlo simulation to accumulate value estimates to guide towards highly rewarding trajectories in the search tree. In other words, MCTS pays more attention to nodes that are more promising, so it avoids having to brute force all possibilities which is impractical to do. At its core, MCTS consists of repeated iterations of 4 steps: selection, expansion, simulation and back-propagation [11]. This process is depicted pictorically in 6

If you think about the entire game of Mendikot, the search tree is unbelievably large, in the order or $10^{15}$ nodes. Hence, our approach was to build a small section of this tree and figure out the best possible action whenever it was the agent's chance to play. So essentially we find the best action that the agent can take in a trick to maximize it's chances of winning the trick, as well as playing the right card if the agent knows that they cannot win the trick.

Each of the nodes is represented by a Node container. This container has all the information necessary to implement MCTS. The Node has attributes

- player: Current player
- state: State of the game
- actions: Cards that can be played in the current state
- parent: The parent nodes of this node
- children: The children of this node
- simulations: The number of times this node was simulated
- returns: The returns obtained from this node

In the selection phase, the agent uses a UCB based policy to select an action that leads towards maximum returns. The UCB value is calculated as:

$$UCB(node) = \frac{node.returns}{node.simulations} + c\sqrt{\frac{\log node.parent.simulations}{node.simulations}}$$

The selection phase continues until the algorithm stumbles upon a node that has not been fully explored yet, meaning that a state in which all the available actions have not been tried. Once the algorithm finds such a node, then the exploration phase starts, and we simply randomly pick a unexplored action

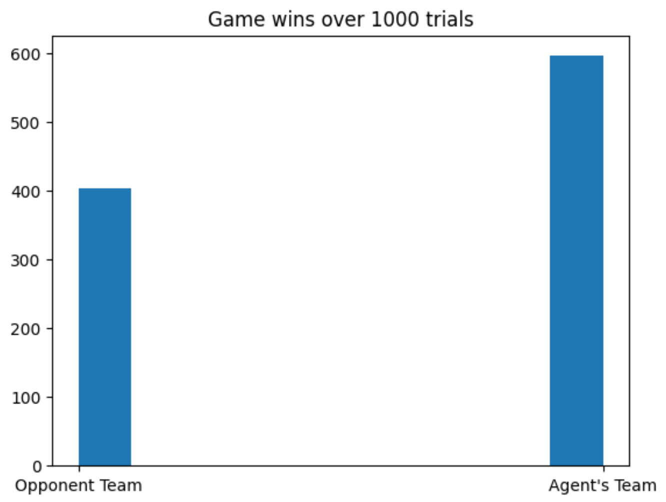


Fig. 7. MCTS overall wins

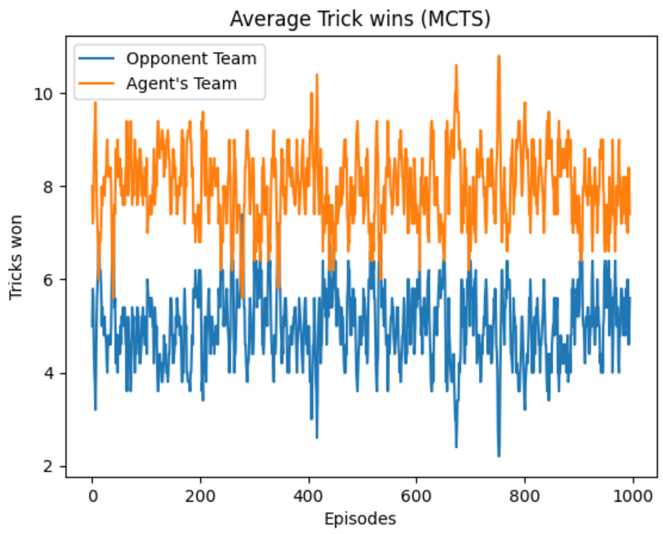


Fig. 8. MCTS average trick wins over 1000 games

leading to a new node. This new node is then attached to the tree, and here the simulation phase begins.

During the simulation step, we roll out one or multiple simulations with reward accumulated for each simulation. Roll-out policy is normally simply or even pure random such that it is fast to execute. This step is like the agent trying to think ahead how the other players might play and what reward he can expect. Once the reward for the trick is obtained, it is backpropagated and the returns are updated.

It takes the agent about 0.5 seconds to roll out about 500 simulations and come up with the best card that can be played in the current trick. The results obtained by using MCTS are shown in 8. The agent's team was able to win an average of 8 tricks out of the total 13 tricks in a game. Overall, the agent's team won $\approx 60$ percent of the total 1000 games.

**Algorithm 1** MCTS: Selection

```
1: Input: root_node (The node containing the state of the
   current trick)
2: curr_node ← root_node
3: while curr_node is fully explored do
4:     for actions in root_node do
5:         ucb[action] ← calculate_ucb(action)
6:     end for
7:     best_action ← argmax(ucb)
8:     curr_node ← curr_node.children[best_action]
9: end while
10: return curr_node
```

### B. Function Approximation

Function approximation is a fundamental concept in artificial intelligence, particularly in the context of reinforcement learning. It involves approximating an unknown function using a simpler, parameterized function that can represent a wide range of inputs and outputs. In

**Algorithm 2** MCTS: Expansion

```
1: Input: node (The unexplored node)
2: if trick is completed in node then
3:     return node
4: else
5:     action ← random choice from node.actions
6:     next_state, reward ← environment.step(action)
7:     new_node ← Node(next_state, reward)
8: end if
9: return new_node
```

**Algorithm 3** MCTS: Simulation

```
1: Input: node (The node from which the simulation starts)
2: reward ← None
3: player ← node.player
4: while reward is None do
5:     actions ← node.actions
6:     action ← random_choice(actions)
7:     reward ← environment.step(action)
8:     player ← environment.next_player(player)
9: end while
10: return reward
```

**Algorithm 4** MCTS: Backpropagation

```
1: Input: simulation_node, reward (reward obtained during
   simulation)
2: node ← simulation_node
3: while node.parent is not None do
4:     node.returns ← node.returns + reward
5:     node ← node.parent
6: end while
```

reinforcement learning, function approximation is often used to estimate value functions or policies in complex environments with large state spaces. By representing these functions using parameterized models such as linear functions, neural networks, or decision trees, agents casn efficiently learn and make decisions based on the observed data. Function approximation enables agents to generalize from limited experience, adapt to new situations, and ultimately improve their performance in challenging tasks.

In the Mendikot enviroment, we identified that the number of unique states and actions were drastucally large in number. Thus, conventional reinforcement learning algorithms like Q-Learning and SARSA would be difficult to implement given their reliance on space-intensive look-up tables.

We decided to implement a function approximation-based approach for determining Q-values for state-action pairs. As discussed earlier, the approximated linear function provides a generalized solution for every state and action. This generalization can be achieved with limited experience which enables us to not run through every possible state and action while training.

The state representation for every state in the game is an array of 9 key features as described above. This array is used as a feature vector. The maximum number of actions possible for the agent are the number of cards that the agent gets when the cards are dealt. The Q-value for a state-action pair can be given as

$$Q(S, A) = w * feature$$

Here, $w$ is a weights matrix which is $n \times m$, where n is the number of cards in play and m is the number of features.

The implementation is documented in the form of pseudo-code as shown in Algorithm 5. The `update` helper function updates the weights array as per the calculated TD error. The `predict` function decides the next action for the agent by choosing the action that gives the maximum output from the approximation equation.

The results for the algorithm were visualized in the form of a histogram of rewards earned for every trick. This helped us gain insights about the winning ratio of the agent. It can be viewed in Figure 9 that the number of times a positive reward is obtained is higher than the negative reward.

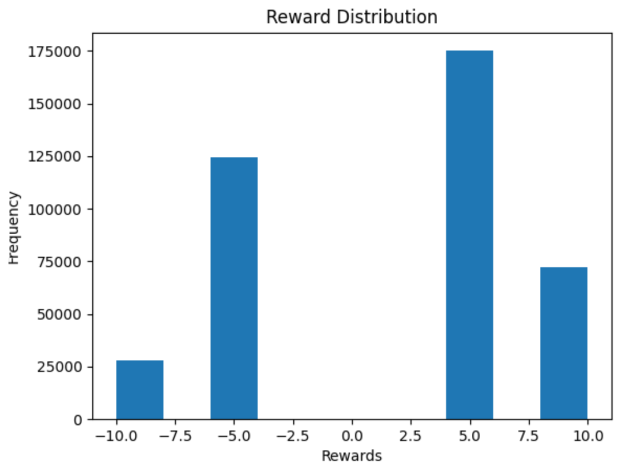


Fig. 9. Reward Distribution using Function Approximation

**Algorithm 5** Function Approximation

```
1: Initialize Rewards Array: *rewards*
2: Define Environment from class *mendikot*
3: Import *MACROS*
4: $state \leftarrow env.reset()$
5: $dict\_tag\_to\_baselink \leftarrow None$
6: for $episode \in num\_episodes$ do
7:     $state \leftarrow env.reset()$
8:     $done \leftarrow False$
9:     while not Done do
10:        if current_player is agent then
11:            action = get_epsilon_greedy_action(state)
12:        else
13:            action = random(available_actions)
14:        end if
15:        next_state, reward, done, winner = env.step_agent(action)
16:        td_target = reward + $\gamma$ * (max(predict(next_state))
17:        update(state, action, agent)
18:        $state \leftarrow next\_state$
19:        Rewards.append(reward)
20:    end while
21: end for
22: Return Rewards, Weights
```

## V. Conclusion

The developed environment was initially tested using 3 approaches as discussed above. We have used the random play results as shown in Figure 4 as baseline metric for comparing the implemented AI-based algorithms. Both the algorithms : MCTS and function approximation performed fairly better in the developed Mendikot environment. It was observed that the agent's winning ratio was significantly better than the random agent implemented in Figure 4. The agent's win percentage was 61% with MCTS and 58% using Function Approximation. Also, from the reward distribution, it was visualized and verified that the positive rewards from total tricks were greater than the negative rewards. This indicated that the number of trick wins (both with 10 and without 10) increased.

## VI. Open Challenges

We anticipate that more sophisticated algorithms such as DQN, and actor-critic methods would be useful for improving the performance of the agent. As discussed earlier, the learned policy would be significantly more optimal if the agent is trained against a team which has a better policy than random. Optimizing the reward function also remains an open challenge. We think that to achieve a human-like strategy, the AI agent should learn to let go off some tricks and preserve higher cards for the next rounds of play. One way to achieve this by implementing a more dynamic reward function that can mold according to the game situation.

## References

[1] Zha, Daochen, et al. ”Rlcard: A toolkit for reinforcement learning in card games.” arXiv preprint arXiv:1910.04376 (2019).

[2] O. Baykal and F. N. Alpaslan, ”Reinforcement Learning in Card Game Environments Using Monte Carlo Methods and Artificial Neural Networks,” 2019 4th International Conference on Computer Science and Engineering (UBMK), Samsun, Turkey, 2019

[3] A. L. Samuel, “Some studies in machine learning using the game of checkers,” IBM Journal of research and development, vol. 3, no. 3, pp. 210–229, 1959.

[4] ——, “Some studies in machine learning using the game of checkers. iirecent progress,” IBM Journal of research and development, vol. 11, no. 6, pp. 601–617, 1967..

[5] ——, “Programming backgammon using self-teaching neural nets,” Artificial Intelligence, vol. 134, no. 1-2, pp. 181–199, 2002.

[6] R. S. Sutton, “Learning to predict by the methods of temporal differences,” Machine learning, vol. 3, no. 1, pp. 9–44, 1988.

[7] Fernandes, Pedro Ricardo Oliveira. ”Framework for Monte Carlo Tree Search-related strategies in Competitive Card Based Games.” (2016).

[8] Daniel Whitehouse, Peter I. Cowling, Edward J. Powley, and Jeff Rollason. Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game. Proc. Artif. Intell. Interact. Digital Entert. Conf., pages 100–106, 2013

[9] Timothy Furtak and Michael Buro. Recursive Monte Carlo Search for Imperfect Information Games. Proc. IEEE Conf. Comput. Intell. Games, pages 225–232, 2013

[10] https://gymnasium.farama.org/index.html

[11] https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168