Card Playing Agent with Minimax and Pruning

Parthan S Olikkal, Saad Rahman, Raghav Deivachilai, Nishanjan Ravin
{polikka1, rahman2, raghavd1, nishanr1}@umbc.edu

***Abstract*— Card playing has been used extensively in the field of Artificial Intelligence for experimenting how well an agent performs with the given cards at hand. This project aims to create an agent that plays the card game "Spades" with minimax approach along with pruning.**

I. INTRODUCTION

Spades is a card trick game that can be played either as a partnership or as a solo game[1]. The sole aim of the game is to collect the same number of tricks as the bid which was placed before the play began. One of the key differences this card game has is that the trump is either decided by the highest bidder or at random, Spades suit being the default. Each player is dealt with thirteen cards in their hand. The game follows in a clockwise pattern and the first player plays the card of his choice and the other must follow the suite if they can; otherwise they may play any card including the Spades suit. Once the card leaves the player hand it is treated as invalid till the next round.

As the states of the game is easy to represent compared to the abstract field of nature, AI researchers are more drawn into the card playing world. The card playing world is usually restricted to a small number of finite actions, as a result the outcomes can be well understood.

In this project, we are considering only Spades as the trump and the total number of players playing the game is at most four. The cards are ranked from highest to lowest with the values, 13 to 1, 13 being Ace and 2 being 1 respectively.

II. MOTIVATION

We started thinking in terms of the different positions our agent would play and apply techniques which would help our agent in choosing the optimal card to play for that position. Considering the large possibilities our agent needs to encounter when it plays first or second, Reinforcement Learning was our firsthand choice, but it did not work out well and our agent was extremely slow in performance. We then shifted to an Expectiminimax approach which would also be favorable in this position.

However, we soon realized that our agent could take a lot of time trying to compute the next best possible move considering the search space. To overcome this challenge and to make our agent significantly faster, we have reduced the lookahead only to the next card and have used heuristic pruning to prune the tree according to the suite and by comparing the highest card in that suite with the cards we have. This enabled us to reduce the search space and choose the optimum card to play in a quick time.

When our agent plays third or fourth, we went with a rule-based approach instead of continuing with the Expectiminimax as the future possibilities are less and we can be sure of a card to play. Our rule-based system keeps track of the cards that the opponents have already played and plays the game accordingly.

III. APPROACH

A. *What is Expectiminimax?*

Expectiminimax is a modified version of the minimax algorithm, a tree-search algorithm which is used extensively in artificial intelligence implementations, mainly in game-playing contexts (the following discussion considers implementation of minimax and expectiminimax in game-playing scenarios only). Each state of the game is considered as a node, and each node has its own utility value (also known as the minimax value), which is a numerical representation of the usefulness (or desirability) of the particular state of the game with respect to the players of the game[2]. The basic idea of the minimax algorithm, for a two-player game (named Max and Min), is as follows:

1. From a particular state of the game (start node), all of the possible moves/options are considered and expanded.
2. The above step is repeated until an end state is reached, where it is now possible to evaluate the desirability of the end state for a particular player. For example, an end state in which the Max player wins can be represented with a positive 1, while an end state where the Min player wins can be represented with a negative 1. An end state ending in a draw for both players can be represented as a 0.
3. The minimax values of the end states are then propagated up through the tree, staying consistent to the idea that the Max player will choose nodes with higher minimax values, while

the Min player will choose nodes with lower minimax values.

4. Thus, with a fully detailed minimax tree, the current player, either Min or Max, will now be able to choose the most optimal move of the game that will result in them having a higher chance to win the game.

While the algorithm is fairly straightforward for a 2-player game, it becomes slightly more complex when considering games with more than 2 players. For such cases, instead of using a single integer value to represent the desirability of the game state to the players, a vector of integers is used, in which the desirability of the game state to all the players is stored. For example, in a 3-player game, an example of the minimax vector at a particular node might be (2, 7, 5). The minimax tree is then filled in by taking the appropriate node in which the player maximizes their minimax value, and propagating that node upwards in the tree. For example, if a current node where player 2 has to make the next move, has 2 child nodes with vector values of (2, 7, 5) and (4, 6, 1), the former vector is propagated upwards, as it has a higher minimax value for the 2nd player than the latter vector.

The above implementation of the minimax algorithm is suitable for games where chance is not involved. However, for games where chance is involved (such as rolling dice, picking a random card, etc.), the minimax algorithm is not applicable, as there is no way of considering all the different possible nodes with their relative probabilities of reaching them. Hence, a modified version of the minimax algorithm, called as the expectiminimax algorithm, is used for such games.

The expectiminimax algorithm includes chance nodes, in which the probability of a particular outcome is considered before it is expanded and listed as a state node from its parent node. These chance nodes include the probability values of the chance action (e.g. 1/6 probability of rolling a fair die and getting a 6, 1/2 probability of flipping a fair coin and getting a "Heads", etc.) in between the parent node and the child node in which the chance action is considered. Hence, when propagating minimax values up the expectiminimax tree, we have to include the values of these probabilities in the calculation as well, before the final minimax value of a parent node can be calculated. For example, if there are two child nodes with minimax values of 2 and 6, and with respective probabilities of 0.1 and 0.9, then the minimax value of the parent node will be equal to $(2 \times 0.1 + 6 \times 0.9) = 5.6$.

The main challenge with the expectiminimax algorithm, is that it will very quickly become intractable for long games, with multiple players, and many chance outcomes. Thus, to improve the performance of the expectiminimax algorithm, there are 2 methods which can be used: pruning, and heuristic evaluation functions.

The idea of pruning is pretty straightforward: the minimax tree search only considers a fraction of the nodes that are a part of the tree, and does not expand the rest of the nodes, thereby pruning the search tree. This is done when there are game states that will never be reached if all the players play optimally, and hence, these game states can be discarded when searching through the minimax tree. This will greatly improve the performance of the algorithm, as a large fraction of the search tree can pruned in the best cases, and hence greatly reducing the number of nodes which needs to be expanded.

The latter method, utilizing heuristic evaluation functions, is used when the depth of the search tree is too large. Thus, instead of expanding nodes all the way to the end state, the algorithm is cut off at a certain point, and an evaluation function is utilized to calculate the approximate minimax value at that particular node. This is done by making use of heuristics, which analyzes the game state at the particular node, and approximating the desirability of the game state to each of the players involved in the game.

Both of these methods can be utilized to greatly simplify the search space of the minimax/expectiminimax algorithm. They also significantly improve the performance of the algorithm, at little to no cost of the effectiveness of the minimax/expectiminimax algorithm[3].

B. Implementation

This card game can be modelled easily as a Minimax tree, assigning the agent as a max player and other 3 as a min player. In that case, the algorithm can get bloated with numerous numbers of unwanted iteration.

To solve this issue, rule-based pruning and heuristic pruning is implemented to make the algorithm faster and improve the performance of the card playing agent. The process and flowchart for the approach of out agent will be described in detail in the following parts.

As, in the card game, there is not surefire or optimal strategy to win a game. So, to make the problem simpler and deal with multiple strategy in the same time, we divided the game into two parts.

1. Early game: When the agent is in 1st or 2nd position to play

2. Late game: When the agent is in 3rd or last position to play

Early game strategies:

When out agent is in the Early game, there is so little information about the game. That means, in the early game, the agent must decide which card it's going to play just by looking in its hand. In this case, formulating a minimax tree is quite expensive in the terms of depth and computation.

So, instead of minimax tree, for early game we formulated some heuristic based rule to abide when the agent plays in an early game scenario. The flowchart is provided for better visualization.

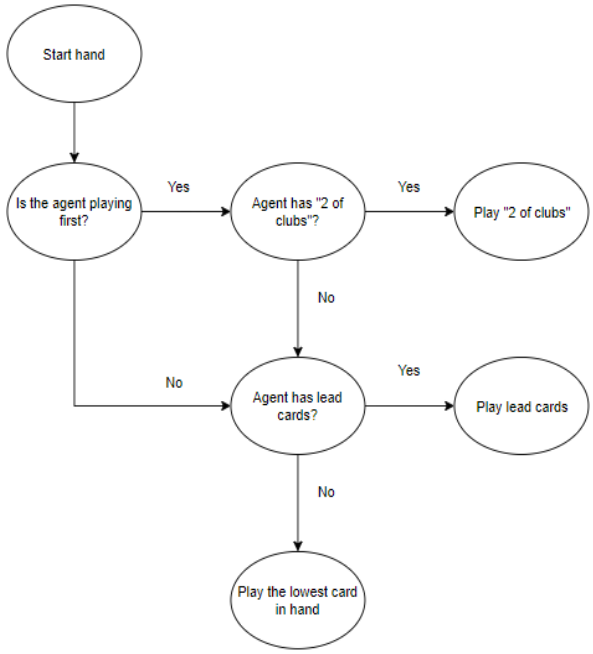

Fig 1: Flowchart for early game strategies

In this case "lead card" means the maximum card of each suite i.e. Ace of Hearts, Spades, Clubs or Diamonds. A tracking algorithm was implemented to keep track of the lead cards during the game. The function used in this part is described in detail below:

TrackingCards:

Tracking Card function shows how the "lead card" of a particular suite is tracked. Every time a new round starts, the entire set of thirteen cards of all the four different suites namely, Spades, Clubs, Hearts and Diamonds are kept in its so called "memory". The highest card of different suites is returned initially as an array. Every time a particular trick is over, tracking card will remove those cards from the initial set. If it's the highest card that is played in a particular trick, it would be removed from the list of cards. This would essentially replace the highest card with the next highest card, thus always keeping track of the highest card. This function is used later to find the best card in the hand depending on the highest card of the suite. Once the entire thirteen tricks are over, then all the suites are empty and when the next round starts, all the cards will be replenished, thereby again keeping track of the highest card. This would continue till the game ends depending on the number of rounds.

Late game strategies:

In the late game, the agent actually can make some intelligent actions based on the information present in the trick. So, in this case, minimax tree with a look ahead of 1 is implemented to select the optimal actions from the available options. The details of the implementation are provided in the following.

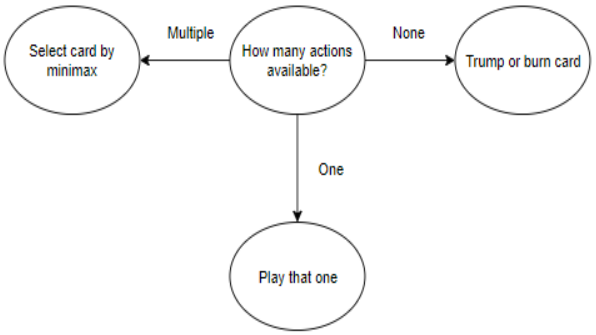

Fig 2: Flowchart for late game strategies

At first, the lead suite (The suit of the first card of a trick) is identified. Then by looking in the hand of the agent, all the valid cards or valid actions for our agent is generated.

Then using the heuristics as Figure 2, the actions were selected according to the number of valid actions the agent had. For example, if the agent has multiple (More than 2) options for a trick, it will select the action by computing the minimax value of all actions. If the agent has only one card, then we don't need to compute the minimax tree. The functions used in this part are as follows:

1. **select_card**: Computes the minimax value of all valid actions of the agent and returns the optimal action(card)
   Input**:** The trick, valid actions of the agent, Approximated opponents hand generated by "trackOpponent"
   Output**:** The optimal action
2. **trackOpponents**: This function tracks the possible cards the opponent might have in their hand. This function would have all the cards excluding the cards at hand.
   Input: Trick suite (what kind of suite)
   Output: Possible cards the opponent might have at their hands
3. **evaluate**: Compares the four cards of the trick and returns 1 if the agent wins the hand and returns 0 otherwise.
   Input: The trick, agent's playing position
   Output: 1 or 0
4. **analyse_hand**: Analyses the agent's hand and returns a dictionary of the number of cards present in all suites with the suites as the key.
   Input: Agent's hand
   Output: Dictionary containing the number of cards.
5. **burn_card**: Generates a possible list of all trumps and burnable cards for the agent. Then

according to the heuristics, return the optimal action(card). The heuristics is presented in Figure 3.

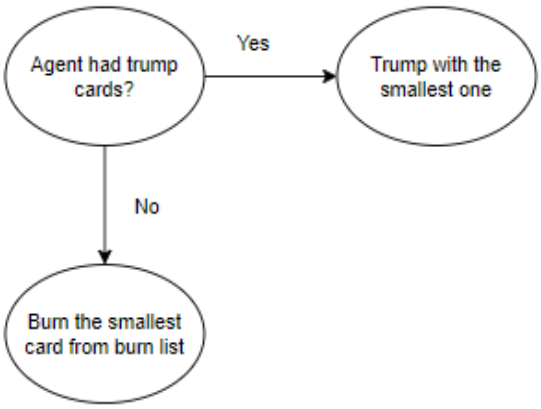

Fig 3: Flowchart for the burn_card function

The "burn list" is a list of possible burn cards. By analyzing the number of cards present in all suites by "analyse_hand" function, the suites having just one card and the suites with having maximum number of cards (without trump suite) was selected for burning. Then the smallest card was returned as selected card. The reason for choosing the suites having just one card is if we burn this card, we can trump the next trick with this suite. The reason for choosing the suite with maximum number of cards is pretty much self-explanatory (More number more spares).

Input: Agent's hand
Output: Agent's selected card

IV. EXPERIMENTS

With the approach at hand along with different functions, we first created a random agent, on the given template for random agent, that outputs a random card. We tested the entire project in Spyder, an open source cross platform integrated development environment in Python 3.7.0. As expected, the random agent was returning a random card upon testing.

The next phase was to replace the randomness that was incorporated in the agent with appropriate cards to play depending on the given hand. Since the card playing game required more than a single player, we tried to create similar replicas of our agent that plays with itself. Although we were not able to fully make the other agents work according to different situations, we learned a lot through the process. With this limitation, we were not able to obtain a fully concrete functional agent as there were different test cases that were yet to be tested which checks if the created agent performed well enough.

With the game engine environment provided for testing the agent with other random agents, we received a win result of 50% once the randomness was removed. It was observed that before implementing trumping over trick feature, the games which the agent had won was won by fine margins. A screenshot of the agent (without trump), "Bluffmaster", running with the game engine is shown below.

```
Game scores: {'Bluffmaster': 102, 'Leonardo': 85, 'Rafael': 101, 'Donatello': 89}
*****
Game Over! Bluffmaster wins!

Final Scores
------------
        Bluffmaster: 102
        Leonardo: 85
        Rafael: 101
        Donatello: 89
```

Fig 4: Screenshot of the agent (without trump), "Bluffmaster", running in the game engine

Furthermore, we ran the agent with the game engine several times to evaluate the performance of the agent, and get a rough idea of how well the agent is performing. We simulated a total of 100 games with the agent, of which the final agent's placings were noted (the agent achieve 1st place in 48 of the 100 games). A graph of the above result was then plotted, which is shown in the figure below.

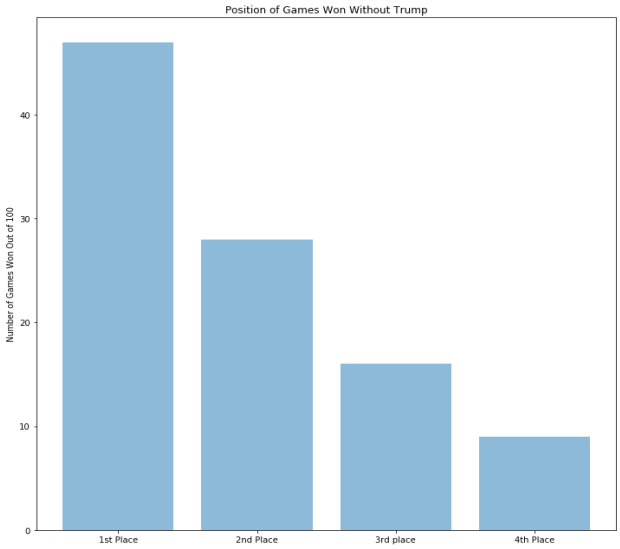

Fig 5: Graph of the results of 100 game simulations (without trump)

We then implemented the trumping over trick feature which improved the way how our agent performed while playing with random agents in the given environment. By trumping over trick, we mean that a particular suite is designed to be higher ranking that any other suite, and also, to play a trump card on a trick[1].

The agent was performing well enough with the other random agents when different test cases were given like the position and cards. The agent performed well when the agent had the highest card and when the agent had trump cards. It was also observed that the agent, not only was winning a higher percentage of games, but it was also winning those games by much larger margins than those observed when implementing the agent without trump. A screenshot of the agent (with trump) running with the game engine is shown below.

```
Game scores: {'Donatello': 65, 'Rafael': 75, 'Bluffmaster': 100, 'Leonardo': 85}
*****
Game Over! Bluffmaster wins!

Final Scores
------------
        Donatello: 65
        Rafael: 75
        Bluffmaster: 100
        Leonardo: 85
```

Fig 5: Screenshot of the agent (with trump), "Bluffmaster", running in the game engine

Once again, we ran the agent with the game engine several times to evaluate the performance of the agent, similar to what we did when we evaluated the agent without trump. We simulated a total of 100 games with the agent, of which the final agent's placings were noted (the agent achieve 1st place in 63 of the 100 games). A graph of the above result was then plotted, which is shown in the figure below.

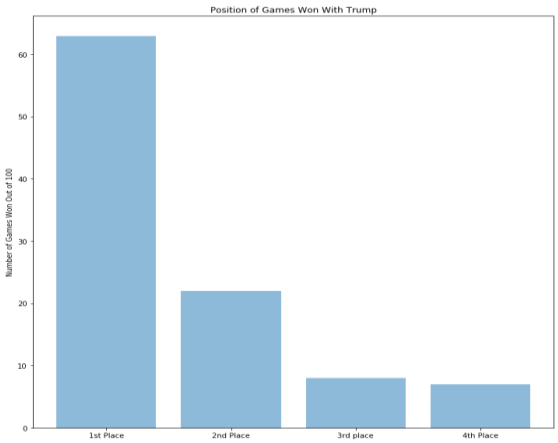

Fig 6: Graph of the results of 100 game simulations (with trump)

V. CONCLUSION AND FUTURE WORKS

With the final agent devoid of randomness and trumping over trick feature, the agent performed with better results compared to previous scenarios. The results were as good as 65% which showed much promise compared to the one with randomness.

In conclusion, our agent showed to perform well in all the test cases that we provided and along with the trumping over trick feature, we were able to get a better performance comparatively.

Creating a tracking of the cards that are remaining of the different suite and the possible cards the other agents were holding were actually a good strategy that we incorporated in the agent. But we should have given more thought and came with a better pruning and rule based approaches when the agent is at the position one and at position two.

We would have looked at a more rule based approach with the tracking strategies that are mentioned above, if given a chance to redo the same differently. And instead of looking only one step ahead, we would have looked at more than one step which would effectively improve the way our agent performed.

REFERENCES

[1] K. A. Schalk, "How to Play Spades," Publications International, Ltd.

[2] A. Katz and E. Ross, "Minimax | Brilliant Math & Science Wiki." .

[3] S. Russell and P. Norvig, Artificial Intelligence. Pearson, 2016.