Jack Amend, Cameron Zach
CIS5603
5/4/2021

Final Report: *Oh, Hell!* Card Playing AI Bot

Introduction

AI programs designed to play games have seen a large amount of publicity in recent years. The program from DeepMinds, AlphaGo, gained notoriety after defeating the Go World Champion, Ke Jie. This program utilized machine learning to beat the champion and used moves that other players described as "alien" [2]. Additionally, in late 2019, Facebook showed off an AI that was capable of playing the game Hanabi [7]. This card game involves human collaboration to get a hand that maximizes the player's points. Their AI was able to significantly outperform human players [9]. Based on the success of other AI game bots in other games, we created an AI program to play the card game, "*Oh, Hell!*".

The objective of the game "*Oh, Hell!"* is to earn the most points by taking tricks. At the beginning of each round, the dealer passes out the cards to each player. The number of cards dealt begins at one for the first round, two the second, and continues to increase for every round until the max number of cards have been dealt. Afterwards, the number of cards decreases by one every round.

Once all players have been dealt their hands, the next card on the deck is flipped and is used to set the trump suit. As in other card games, the trump suit sets the value of any card of the same suit higher than every other card of a different suit. For example, if the trump suit is spades, then every card whose suit is spades is higher than any card with a suit of either hearts, clubs, or diamonds.

After the reveal of the trump suit, players then bid on the number of tricks they believe they can take this round based on the cards in their hand. The player to the left of the dealer lays down one card first and then the players follow in a circular fashion. When going around, players must follow the leading card's suit, but if they do not have any cards of that suit, they may play a card of any suit. The player who laid down the highest card takes the trick and then lays down the first card for the next trick. Once all the cards have been played, points are counted; each trick is worth one point, and if a player meets their bid they receive a bonus 10 points.

Background

Recent research has shown success in AI playing games better than humans. There are some intuitions that we believe explains why AIs have the potential to play this game better than humans. Firstly, the AI has an ability to perform more complex computations than the human players. This can potentially be leveraged to make highly accurate predictions on the number of tricks the AI expects to take. This can be seen in a similar AI, designed to play poker. In [1], they designed an AI to play Texas Hold'em Showdown, and their AI outperformed all the human players. Additionally, an AI for the game "*Oh Hell!*" can keep a better log of what cards have already been played in the round to make informed decisions on which cards to lay down during

each trick. While the rules of the game are somewhat straightforward, we thought the probabilistic reasoning required would pose an interesting challenge. The environment is only partially known to the AI at the start of the round; the AI knows what is in its hand and what the suit and value for the trump card is. As more tricks are played, the AI gets to observe more of the environment by seeing what cards are being played to then make better decisions about what card to play next.

A similar application for an AI program is presented in [8], where the author created a program to play the German card game, "*Skat*". Like "*Oh, Hell*", this is a trick-taking game. "*Skat*" is played amongst three people and a 32-card deck. The authors of this paper proposed a new way of sampling to boost the performance of their search algorithm. That is, they use the history of moves in order to sample more realistic potential future states.

In this project, we implemented two different state search algorithms, a simple tree tree search algorithm and the Monte Carlo Tree Search algorithm, the latter being one of the driving engines behind AlphaGo [4]. These techniques have been used in other AI game applications. In the paper, [3], the authors use a neural network to learn how to search better using Monte Carlo Tree Search. As is similar to [8], the focus of their work is to capitalize on the power of Monte Carlo Tree Search by improving the future state searching. This differs from our project since we are looking to compare search algorithms, rather than just focus on improving Monte Carlo Tree Search.

Another paper also looks at using neural networks for card games. The authors of [6] use two types of frameworks to create an AI card playing program. The first used temporal difference learning and self-play to create a competitive neural network that could perform well against human players. The other used an evolutionary approach – they put two networks against each other and whichever network won, the other network had its weights adjusted 5% towards the winner. They found that this simple algorithm ended up out performing the other approach. Their methodology focused on deep learning and reinforcement learning.

For our work, we implemented a simple tree search algorithm and Monte Carlo Tree Search algorithm and compared them in order to see which performs best. We also created a simple agent that randomly selects valid cards to play. Our objective in this project was to determine which of these algorithms works best and then to evaluate their performance against human players.

Methodology

The simple tree search (STS) was originally intended to be similar to the Alpha Beta search, but ended up being closer to the Minimax algorithm. On the agent's turn, the AI calculates the probability of winning the trick by playing each card in its hand. It makes a very rough approximation of this probability, simply taking the probability that any random card in the deck will be higher than it. To continue down the tree and calculate probabilities for future tricks, the AI needs to know the lead suit for each of those future tricks. Since those suits cannot be known with certainty, we take each of the four suits into account and consider them all to be equally likely. For each potential leading suit, we run the same probabilistic analysis. When we have reached either the bottom of the tree or the max depth parameter, we sum each branch of

the tree from bottom to top. By treating each node in the tree as an indicator variable, we can see that the probability value represents the expected number of tricks won by playing the given card, and by extension, the branch total represents the expected number of tricks the AI can take by playing all of the cards along this branch. This AI will always bid its entire hand size, so maximizing the tricks taken is an optimal goal.

The Monte Carlo Tree Search is an efficient state space searching algorithm. It comprises 4 different steps: selection, expansion, simulation, and backpropagation. To begin, possible states are represented as nodes in a tree. A card being played transitions represents a link from one node to another. Each node contains information on the current game state as well as two other variables. One variable counts the number of wins the given node is involved and the other variable counts the number of times the node has been used in simulations.

To begin the MCTS algorithm, the selection algorithm is performed in which the node with the most potential is identified to explore. This is implemented following the proposed heuristic in [5]. The Upper Confidence Bounds algorithm is adapted for trees (UCT) and balances between exploitation and exploration when deciding the best node to explore. Once a node is selected, the expansion phase begins. In our implementation, a single possible action is expanded from the selected node and appended to its list of children. After this, the simulation phase runs until a final state is found. During the simulation, we implemented a random policy for making moves. During this stage, no new states are added into memory, but instead are used to reach a terminal state. The backpropagation stage then begins by updating the tracking variable for each node in memory that is on the path from the found final state to the root node. Every node in the path has its variable that tracks the number of simulations increased by one. If the final state resulted in a win for the agent, then the variable corresponding to the number of wins is also increased.

To interact with the agents, we developed an interactive web interface where users can play the game against our AI. This application allowed us to visualize the game play and make it easier for people to interact with, while also providing a much friendlier interface than just numerical outputs. The web app is written with React, and communicates with a backend Python script via running socket-io. Upon connecting, users can configure the settings for the three AI players, and then can play a full game of *Oh, Hell!* Both the frontend and backend are hosted on Heroku, and are publicly accessible at oh-hell-frontend.herokuapp.com.

Results

To test the AI agents, we ran 5 experiments. For the two search-based AI agents, we experimented with all four players using the same algorithm with different parameters. We then also put two players using the same search algorithm against two players playing randomly. In the last experiment, we used two MCTS players and two STS players. Each experiment ran 500 games and then average points obtained were calculated and recorded.

The STS AI performed quite poorly. The experiment with four STS AI showed that exploring beyond one node deep in the game tree only reduces performance. This was a surprising result, but it was likely due to the fact that so much of the information in the game is hidden. We hypothesize that by exploring the tree too far, the AI is giving too much consideration to game states that are impossible given the current hidden information. We originally thought that the AI would be able to overcome the problem of hidden information

based upon it making probabilistic decisions, but ultimately it seems that the AI agent became trapped in looking at bad and unlikely future states.

When matched against players who make completely random decisions, the failings of the STS agent become apparent. On average, the best performing STS AI managed to score just 17 points per game, while the AIs making completely random choices managed to earn 44 points each. This evidence leads us to believe that the algorithm implemented for the STS AI is not sufficient to play the game at any level of human proficiency.

The MCTS algorithm fared much better. Interestingly, it saw a similar decrease in performance as the search time increased. This is likely due to the same issue as before; it is impossible to accurately predict the next game state with most of the state information being hidden. Thus, compounding too many uncertain predictions results in a substantial decrease in performance. When playing against the random AI agent, the results were somewhat surprising. The MCTS players scored an average of 46 points, but the random players also scored an average of 42. The MCTS algorithm is used in many applications, but we found that it only slightly outperformed a completely random agent. We believe this occurred because the decision making policy was random, unlike previously discussed research that heavily relied on machine learning. Unlike other implementations of MCTS for games, we did not use any reinforcement learning. This could be why this algorithm did not perform as well as expected.

In the final experiment, the MCTS algorithm easily beat out the STS algorithm, earning an average of 47 points per game compared to the 17 points earned by the STS agent. Since the STS AI proved to be so terrible at this game, this experiment did not lead to as much insight as we originally expected. The margin by the MCTS algorithm beat the AlphaBeta algorithm is similar to that of the random player, but based on average scores alone it is hard to say why that happened.

To gain a more in depth insight into the AI agents' behaviors, we each played several games against each opponent. Cameron found that the completely random agent performs surprisingly well, because it reliably scores 0 points and can get the free 10 points for betting 0. Playing a random strategy himself proved to be extremely effective. This explains why the random AI performed so well when paired with MCTS. The better AI was able to take tricks more reliably, allowing the random player to more frequently hit its most common bid of 0. STS performs poorly in this scenario because the random AI will occasionally take a trick, which ruins the STS bid of its entire hand. This trivial strategy wasn't taken into account when designing the AI.

Cameron - game 1					**Cameron - game 2**				
Round	**North (MCTS 1s)**	**South (Human)**	**East (STS 1)**	**West (Random)**	**Round**	**North (MCTS 1s)**	**South (Human)**	**East (STS 1)**	**West (Random)**
1	1	10	0	10	1	11	10	0	10
2	2	21	0	20	2	12	20	1	20
3	3	22	0	31	3	13	30	3	20
4	13	24	2	41	4	13	40	17	20

5	14	24	3	44	5	24	42	19	20
6	15	27	4	55	6	24	45	20	22
7	18	30	5	55	7	26	55	23	24
8	19	33	5	57	8	28	56	26	34
9	20	35	6	68	9	28	66	28	37
10	32	47	6	68	10	31	67	28	37
11	42	47	7	80	11	33	68	28	47
12	52	47	19	80	12	34	78	29	57
13	62	57	30	90	13	34	79	29	67

Table 1: Scoreboard from two matches of Cameron playing against the AI agents.

Jack played against the AI agents with an aggressive strategy. During these trials, the MCTS and Random agents performed fairly well but STS was unable to match their performance. During his first match, he was in the lead for the entire game. At one point, the MCTS algorithm was only two points behind, but ultimately failed to surpass him, as can be seen in the left sub-table of Table 2. We believe this is due to how the MCTS defines success and thus prioritizes which states to look for. MCTS is rewarded for reaching a success state in which the number of tricks it has taken is equal to the number it originally bid. However, if the agent begins where this is already impossible (for example, the agent bids 0 but has 1 trick), then no matter what search path they take, it will result in a failure. This means the agent will consider all the options to be equally bad and randomly pick one. A different strategy, when in this situation, could be to attempt to take as many tricks as possible to get more points since the 10 point bonus is unattainable. In Jack's second match, the AI agents began with a steady lead. It was not until the third-to-last round that he was able to gain a lead over the AI agents. While ultimately winning again, it was surprising to see that the agents were able to perform decently well against a more experienced player.

Jack - game 1					**Jack - game 2**				
Round	**North (MCTS 1s)**	**South (Human)**	**East (STS 1)**	**West (Random)**	**Round**	**North (MCTS 1s)**	**South (Human)**	**East (STS 1)**	**West (Random)**
1	1	10	0	10	1	10	0	11	10
2	1	11	1	10	2	21	1	11	20
3	4	21	1	10	3	31	11	12	22
4	14	33	1	12	4	42	13	12	23
5	26	35	2	22	5	42	25	14	34
6	38	47	4	32	6	42	36	16	47

7	40	49	6	43	7	44	36	18	50
8	41	52	8	43	8	46	39	18	51
9	42	53	11	43	9	47	41	18	53
10	52	63	12	46	10	58	52	19	54
11	62	64	12	48	11	58	63	20	55
12	63	74	12	59	12	59	73	21	65
13	64	84	12	69	13	69	83	32	75

Table 2: Scoreboard from two matches of Jack playing against the AI agents.

Conclusion

Our results present some interesting findings. Firstly, the trend that deeper and longer searches result in worse performance for both AI agents. When increasing the STS depth, the performance decreases. The same phenomenon occurs when increasing the search time allowed for the MCTS. It seems counterintuitive that searching more would cause a decrease in performance. We postulate that this occurs because the game only has a limited amount of information available. Looking too far into the future allows for the consideration of too many options and the decisions become closer to random, if not worse. We also attribute this to the agent decision policy. We did not implement any reinforcement learning so the model is limited to simply searching for any possible outcomes. MCTS seems to perform better than STS since it is able to prioritize better nodes to explore.

We found implementing the web application was invaluable to the project. Firstly, it made interacting with the agents much easier. We were able to see the decisions being made by the agents in real time to gain a better understanding of how they are performing. Secondly, it increased the enjoyability of playing this game. The interface made it more like actually playing this card game.

Our human experiments using the web application were some of the most valuable pieces of our analysis. We were able to see how the AI would respond to different styles of play, and closely examine their round-to-round performance. Ultimately, we concluded that even our best AI, the MCTS player, still needs improvements before it is able to reliably defeat human players. Though it is able to perform close to a human level in some cases, during all four tests, there were games where the random strategy defeated the MCTS agent.A stronger AI would recognize this random player's frequent zero-bidding and attempt to thwart it by intentionally losing a trick. Performing these calculations would explode the search tree very rapidly, so in order to improve the AI in this area we believe that some form of reinforcement learning would be necessary.

In future works, we would like to experiment with other ways to improve our AIs. Specifically, we would like to use reinforcement learning to create AI agents that can outperform any other player, be them AI or human. Additionally, we would like to see if there is a way to fix the problem of decreasing quality when increasing the searching. Potentially, for the simple tree search, a different heuristic could be implemented. For MCTS, a different way of simulating the games could be beneficial. Currently, the policy in the simulation step of the MCTS algorithm is a random card is chosen to play until an end state is reached. We would like to add in some additional logic for making decisions during the simulations. This could potentially improve the quality of longer searches to make them more usable.

References:

[1] A. Blair, A. Saffidine, AI surpasses humans at six-player poker, Science Magazine, Aug 2019.
[2] D. Chan, The AI That Has Nothing to Learn From Humans. *The Atlantic*, October 2017. https://www.theatlantic.com/technology/archive/2017/10/alphago-zero-the-ai-that-taught-itself-go/543450/
[3] A. Guez, T. Weber, I. Antonoglou, K. Simonyan, O. Vinyals, D. Wierstra, R. Munos, D. Silver, "Learning to Search with MCTSnets", ICML 2018.
[4] M. C. Fu, AlphaGo and Monte Carlo tree search: The simulation optimization perspective, *2016 Winter Simulation Conference (WSC)*, 2016.
[5] L. Kocsis and C.Szepesvári, Bandit based Monte-Carlo Planning, European conference on machine learning, 2006.
[6] C. Kotnik, J. Kalita, The Significance of Temporal-Difference Learning in Self-Play Training TD-rummy versus EVO-rummy, ICML 2003.
[7] A. Lerer, H. Hu, J. Foerster, and N. Brown, Improving Policies via Search in Cooperative Partially Observable Games, AAAI 2020.
[8] C. Solinas, D. Rebstock, M. Buro, Improving Search with Supervised Learning in Trick-Based Card Games, AAI 2019.
[9] Q. Wong. Facebook's new card-playing bot shows AI can work with others. *CNET*, December 2019. https://www.theatlantic.com/technology/archive/2017/10/alphago-zero-the-ai-that-taught-itself-go/543450/