Application of Expectimax For a UNO Agent

Andrew Guerra
guerr242@umn.edu
The University of Minnesota
Minneapolis, Minnesota, USA

Grace Condie
condi031@umn.edu
The University of Minnesota
Minneapolis, Minnesota, USA

May 9, 2023

1 Abstract

Thousands of agents have been created over the years to compete with a number of player to win a variety of different games. To do this, they have used a variety of algorithms and problem solving techniques to create artificial intelligence that is able to out-perform other types of agents in a range of situations. With that said, within this research, we will report on one game in particular, UNO, and how an agent was implemented using the Expectimax algorithm in order to achieve the goal of successfully winning games of UNO at a higher probability than a random chance agent. This report dives into the implementation of said agent and how it works in tandem with the Expectimax algorithm to play a game of UNO. By the end of the report, after finding and analyzing the results from running experiments on the agent in different situations, it was found that the Expectimax agent only had a slight advantage over the random agent(s). It also only resulted in a small decrease in the number of moves within a game, demonstrating that the agent as a whole did not have a significant impact on win percentage or on the overall efficiency of play. Once this conclusion is reached, the report also addresses some of the shortcomings and bias of the agent's testing, as well as future work that may resume in this area of study related to agents for stochastic games.

2 The Problem

The goal for this proposal is to successfully create an agent that can win multiple games of UNO at a higher probability than a random chance agent, following the definitions elaborated on below. Within the next sections, the rules of the game, the definition of the agent's environment, and the appeal of the problem itself are further addressed.

2.1 Rules of the Game

UNO is a multiplayer game that is played with at least two and up to ten players [6].

Each player is initially dealt seven cards and one card is revealed from the draw pile and put in the discard pile. Each card has an associated color, with there being four possible colors per card. During a player's turn, they can play a card that corresponds to either the color or number of the card on the top of the discard pile. If a player cannot play a card during their turn, they must draw a card from the draw pile. If a draw card is playable, it must be played.

Number Card	A number card is the standard card in UNO. It has a number value from zero to nine.
Reverse Card	A reverse card reverses the order in which players play. For instance if the player to the left of the current player is set to play next and a reverse card is played, the player on the current player's right will play next.
Skip Card	A skip card will skip the next player's turn and the player after them will play after the current player.
Wild Card	A wild card can be played on any other card. As it is played, the current player will set the apparent value of its color.
Draw 2 Card	A draw 2 card will forces the next player to draw two cards and skip their turn.
Wild Draw 4 Card	A draw wild 4 card can be played on any card and will force the next player to draw four cards and skip their turn. The current player must not have any other playable cards in order to play this card.

The objective of the game is to be the first player to have zero cards in their hand at the end of their turn.

A Wild Card or Wild Draw 4 Card cannot be considered a starting card. If either is drawn as the first draw pile card, it is reshuffled into the deck and another card is draw to the discard pile until the condition is met. A Wild Draw 4 Card can only be played when a player has no other options to play besides that card, excluding additional Wild Draw 4 Cards.

2.2 The Environment

In order to successfully start implementing the AI agent to solve this problem, it is helpful to define the environment

that the agent will be working in. UNO, due to being a game revolved around chance and having multiple players, is in a non-deterministic and multi-agent environment. It is also partially observable due to the fact that the player does not know what card is on the top of the deck or what the other player has in their hand. Lastly, the game environment is both discrete and static, due to no changes being made in the game while the agent is thinking as well as the game not taking place within real world space and time.

2.3 The Appeal

What makes this particular problem and approach so interesting is that the game of UNO is mainly based in chance, making it not very straightforward to win and determine the path to do so. It is also a game that introduces a lot of different elements in it, as well as a game that most people are familiar with and have played before. Due to having experience with the rules of this game definitely added appeal, however having a million different variations in which you can play a single hand makes it an even more exciting project to figure out what is theoretically the best possible tactic in winning the most possible games.

Additionally, it was also appealing due to its application. Along with other UNO variations, other games in a similar stochastic vain as UNO could use the same approach to define an agent. This makes the definition of the agent much more appealing to understand and apply, not only in this research but in other contexts as well.

3 Background

3.1 Introduction

This literature review will discuss a variety of sources elaborating on the possible algorithms and procedures that could help construct an artificial intelligence agent to play UNO. The sources will be presented in thematic order, grouped into categories based on the algorithm that they present as a possible way to go about constructing agents for similar games. These algorithms and procedures include defining the state space of the agent, as well as using Reinforcement Learning, Expectimax and Minimax search, and Markov Decision processes to inform on possible ways to implement such an agent.

3.2 State of the Art

3.2.1 Defining a State Space.

Before even diving into the construction and implementation of an artificial intelligence agent, it is essential first to define the state space, as well as the general rules that the agent has to follow. The article "A Card Description Language"[3] dived into this aspect of state space definition for a variety of different card games, including UNO, and elaborated on using game descriptive language to construct rules that can serve specific purposes, especially in computational analysis. In the definition of card games specifically, they stated a few different axioms that must be considered. These included the number of players, the location(s) where a card is able to be placed during gameplay, what a stage consists of and when it has ended, as well as many others. In addition to these, it encouraged the definition of both mandatory, optional, and play-only once rules in order to make it exceptionally clear to the agent what it needs to factor into its decisions.

Not only is defining the state space itself important, but it is also essential to be able to characterize games and their landscapes. In the article published by Nature Communications titled "Navigating the Landscape of Multiplayer Games"[10], it dived into just that, helping to create a framework for characterizing certain features and rules for games within AI agents. This article discussed the tools that are needed in order to navigate and build these landscapes in the AI field, using a variety of methods such as response graphs and the calculation of computational complexity. With that said, the primary purpose of this article was to dive into some technical characterizations that aid in defining a state space to ensure the best functionality when it comes time to construct the agent itself.

Frameworks such as TAG have also been developed to both setup state spaces for games and modify states in a model of the games [5]. These frameworks allow for a generalized approach to state definition of games of various complexity, allowing researchers as well as programmers to develop models of games as well as agents to operate on said models.

3.2.2 Reinforcement and Deep Learning.

Both reinforcement and deep learning are introduced as a common procedure for winning games similar to and including UNO. Researchers at Stanford University[1] worked closely with deep learning techniques, including Q-learning and SARSA when working to create a successful UNO playing agent. Through their research, they elaborated on the complexity and uncertainty that comes with the game, and how to set up an environment in a way such that the large state space and complexity will be accounted for. In the case of UNO, they stated that both reinforcement and deep learning are ideal agents for similar types of card games, specifically in the case of Q-learning and SARSA. That is due to the technique's ability to reduce the complexity when faced with a multi-agent card game, allowing less interaction with the environment and an overall better game playing agent.

Additionally, in an article titled "Tackling the UNO Card Game with Reinforcement Learning"[11], author Bernhard Pfann elaborated on using Reinforcement Learning to construct an agent that can engage in a game of UNO. In this article in particular, the reinforcement learning techniques that were applied were both Monte Carlo and Q-Learning, both which would prove extremely useful in creating a UNO agent. It first illustrated how to best define the environment

of the agent prior to adding its ability to implement Reinforcement Learning and make a decision in gameplay. It is inherently simple in its approach as you wish to award beneficial moves that can lead to a desirable terminal state as a result of the actions the agent chooses to make. Monte Carlo specifically, however, allows the agent to make decisions by simulating different games and decisions, making it particularly helpful in reinforcement learning and as a whole.

Continuing on the trend of Monte Carlo, authors Lipin, Tang, and Worthy[8] detailed using Monte Carlo Tree Search (MCTS) for the game Durak, which although different from UNO, is also a partially observable and non-deterministic game environment. Throughout the research, they noted that a large amount of computation required slows the agent considerably, which can and will occur when playing games such as UNO, which is definitely important to consider when looking to implement the Monte Carlo algorithm. They also noted that the limited knowledge of the AI makes it so that the agent can still lose, even when playing "optimally". However, even with these limitations, Monte Carlo Tree Search can help in the creation of an AI agent capable of playing similar games to Durak against human players, which makes it extremely useful. It implements MCTS by using a strategy model that allows for a variety of different actions available for agents to choose from. Through the use of these trees and the visibility between players, the agent is able to use the partially visible information to make its best prediction, making it a great approach even with some of its limitations.

Furthermore, another source [15] discussed using reinforcement learning in a variety of card game environments, such as collaborative environments, environments with many potential actions, and environments with large state spaces, including UNO. It proves exceptionally helpful in agents that need to learn goal-oriented tasks, where it has the capability to make decisions on which move to make given certain situations. When it comes to UNO specifically, it touched on the size of the state space in the average game, being approximately 10^{163} states, which is extremely large. Although it is definitely a complex problem to handle, it later went into how Reinforcement Learning techniques can be beneficial in these types of environments. With that said, this article detailed a toolkit by the name of RLCard, which allows for the easy implementation of reinforcement learning in card games without having to start from scratch. This tool proves especially useful in these environments and can definitely be applied to games like UNO.

3.2.3 Minimax and Expectimax Search.

Another form of tree search of game stats is the minimax search algorithm. The Minimax search algorithm models a zero sum game between two players, Min and Max [13]. The search consists of a series of state nodes connected by actions that can be taken by the current player. The player will traverse the state space and will choose actions that maximize or minimize the cumulative value of a set of actions taken [14]. As the branching factor of the state space increases, such as in games with many potential moves, the run-time of the Minimax will also increase. Two possible ways to mitigate this are depth limiting and Alpha-Beta pruning.

Depth limiting of the Minimax algorithm is the stopping of the execution of the algorithm before it reaches all terminal states of the state space on possible branches, meaning the algorithm has not fully explored the consequences of potential moves. In the typical running of the Minimax algorithm, terminal states in the state space provide the inherent value of the move sequence used to arrive at the terminal state [14]. If the depth is limited, an additional heuristic function will be needed to determine the value of any given state in place of the heuristic defined for a terminal state. With this general state heuristic and limiting to a specific branch depth, we can avoid exploring large portions of the state space, increasing the efficiency of the algorithm. However, the defined state heuristic can easily over or underestimate the value of a given state if not carefully designed [13], leading the algorithm to make less advantageous than one that is not depth limited.

Alpha-Beta pruning is the process of eliminating the need to explore branches during the running of the Minimax algorithm due to conclusions that can be made on the actions of an opponent [14]. The best possible value of a branch is kept track of for both opponents Min and Max, with these values being called alpha and beta. The algorithm will then use these values to determine whether a branch can be "pruned" based on if it is considered not better than a given alpha or not worse than a given beta, with the ordering of the previous statement being based on the opponent's perspective. Like depth limiting, this technique can eliminate large portions of the state space that needs to be explored. However, unlike depth limiting, the elimination of branches is based on the value of branches, or lack there of in comparison to the most valuable branch. Depth limiting eliminate branches without considering their value.

Expectimax operates in a similar way to Minimax, minimizing or maximizing a value of a branch, however, this value is determined as the expected value of a branch [14]. Expectimax is used to operate in non-deterministic environments where the probability of a given action happening is known. Given a key feature of UNO is that the state space is defined randomly, this feature is key to making advantageous decisions.

3.2.4 Markov Decision Process.

A Markov Decision Process (MDP) allows the modeling of stochastic environments to make decisions. In partially observable environments, Partially Obervable Markov Decision Processes are used (POMDP). This model is based on a series

of sets: S (the state space), O (the observation space), and A (the action space). These states can be used to generate a reward r, which along with the probability distribution of the belief state, can be used to generate advantageous decisions for the agent [9].

Reinforcement learning and POMDP can be used in conjunction to make powerful agents for large partially observable state spaces [4]. A reinforced learning model can be used to help calculate the reward in a POMDP. A large benefit of POMDPs is their ability to model inference on states, enabling limiting of the search space as well as more effective rewarding of acions [7].

3.3 Conclusion

Working in very large state spaces such as UNO is often noted as a difficult problem for traditional tree search algorithms such as Minimax. In most cases, a reinforcement learning approach is taken in order to take such state spaces. This ultimately can be narrowed down to a duality of needs: time spent to make a decision and time taken to set up the decision maker. Reinforcement learning often takes a long time to train an agent, however this long training leads to a very short time for decision-making. In contrast, tree search algorithms need no such training and setup, but take substantially longer to evaluate each individual decision.

It can be argued that over time, the reinforcement learning approach will save time due to the continual summation of large time steps of evaluation for the tree search approach. However, it is still crucial to note the benefits of a tree search approach.

The tree search approach allows complete containerization of the algorithm. Unlike reinforcement learning techniques, no external information is required, besides the current state, for the running of tree search algorithms. This could be useful in systems where memory is limited. The tree search approach also allows the programmer a complete understanding of the workings of the decision-making process. Reinforcement learning and deep learning approaches can completely hide the notion of how a decision is made, creating a black box the programmer cannot analyze.

4 The Approach

4.1 The Plan

To create the AI agent that is successfully able to solve this problem there are a few logistics that must first be addressed. The software will be written in Python and will consist of the following main components: an UNO game class to represent the state and running of a UNO game, an UNO agent class to represent an agent and its behavior, and an experiment class to evaluate the agent.

The agent will consist of a mini-max implementation [12]. However, due to the UNO environment being non-deterministic, we will implement an evolution of mini-max known as expectiminimax, which is further explained in the section below. [2]. The agent will keep track of the environment as it is visible to the agent, meaning the agent's hand and any cards played to the discard pile. It will then work in tandem with the UNO game class to start a game with the Expectimax UNO agent as one of the players. Finally, to evaluate the agent, the experiment class will multiple iterations of the UNO game with a variety of different agents, resulting in statistics regarding the win percentage of the newly defined agent, working to solve the problem.

To implement different aspects of the agent we plan to use the aima-python code repository, specifically due to a lot of previously implemented functionality that it contains regarding games. This repository particularly helped with keeping track of and changing game states throughout the game, as well as allowing multiple players to take part, and being able to replicate certain aspects of the game such as drawing a card from the deck.

Lastly, it is essential to plan for how the solution is to be evaluated so that it is clear when a solution is achieved. A solution to this problem is achieved if the agent is the first player to have zero cards in their hand. The success of the agent can be evaluated through putting it through extensive testing covering an assortment of situations varying in characteristics such as number of players and the order in which the agent plays certain cards. The program will then keep track of how many games the agent reaches a solution, using it to keep track of its win percentage in the cases of different types of situations as well as overall. This will allow for the decision of whether or not the agent is successful to be made clear. This overarching method to test with is loosely based off of the process elaborated on in the Stanford study "Winning Uno With Reinforcement Learning"[1] focusing specifically on evaluating through an average win rate.

4.2 Expectimax

After the extensive research that was conducted above, it was decided that the best approach to the UNO agent implementation was through an Expectimax algorithm. The thought process in doing so is mainly due to this game theory algorithm being perfect to maximize the expected utility. Since UNO is a non-deterministic environment and therefore very much a game of chance, this was the perfect method to implement this aspect of the game. However, there were some concerns about how well the Expectimax algorithm would work. Since UNO has such a large game tree due to large belief states, the run time of such an algorithm can increase massively with depth, taking too long to make moves even with a depth of only 3 or so branches. With that said, it was necessary to specify a depth limit for the algorithm for the sake of efficiency, which unfortunately did slightly decrease the performance of the agent.

With that said, the part that Expectimax plays in the implementation of the agent is in tandem with the evaluation function, which partakes in obtaining the performance and utility of the agent to inform it of the best play. The algorithm itself however was not implemented directly in the agent code base but instead by using the aima-python repository. Through the use of previously defined alpha-beta-cutoff-search, it is possible to replicate Expectimax within the game, allowing the agent to utilize the aspects of Expectimax that were elaborated on above to perform its best.

5 Experimentation

After choosing the algorithm and planning out the best course of action above, the implementation was ready to begin. In the sections below, both the design and implementation of the code files for the agent will be described, as well as how we implemented the evaluation function and obtained the final results.

5.1 Design

The code is divided into three main files, UNO.py, UNO-Agent.py, and experiment.py, with the implementation of the game board in the first, the agent in the second, and the testing of the agent in the last. All of these files reference the aima-python code repository, which contains a collection of previously defined game theory algorithms as well as game elements that can be utilized. They will each individually be explained below, however, it is important to understand how the files work together. The agent class creates an AI player to partake in a game of UNO. When an instance of the UNO game calls the play-game function, that agent will partake against other players in the game. The Experiment file then comes into play by using a said agent and the ability to play games to simultaneously run a variety of situations within the game to evaluate the performance of the agent itself.

5.1.1 UNO.py.

The design of UNO.py mainly consists of the UNO game class, particularly the game board itself. This is where a lot of the rules specific to UNO are defined and the game is initialized. Some important functionality that needed to be addressed within this class is how the game board should be initialized given certain parameters such as the number of players and the number of cards that a player should start with their hand. It should then set up important functions to keep track of the actions that are possible at any given time, the results of making said action and a way to calculate the utility value of certain game tree nodes. With all that being handled in the design, there must be some way to interact with and begin a game given certain circumstances.

In this game class, the state is defined as a series of hands of cards. One for the draw and discard pile respectfully, as well as for each player in the game. The game will be able to generate new states based on an action applied to that state. An action, card, can only be applied to a state if it is playable as defined by the rules of UNO and if it belongs to a player's hand on their move.

With that said, the following table depicts the main functions from the UNO class and how they are utilized to build the game.

generate-initial-board	This function initializes the state via the board and hands of the players. All cards are generated for the standard UNO deck [6] and are shuffled. Each player is dealt 7 cards and a card is dealt to the discard pile to start the game.
actions	This function returns a list of playable cards for the current player based on the card on the top of the discard pile. Wild cards are considered playable regardless of their assigned color.
result	This function generates a new state based on a card action on the current state. This function is responsible for reversing the direction of play in the game as well as who the next player is as well.
utility	This function returns the value of the terminal state based on who the player at that state. The utility will be positive if the first player is the winner of the game.
display	This function prints out information about the current game state: draw pile, discard pile, and all the contents of the players' hands.
play-game	This function plays the game as it was defined with n players. It will return the player that has won along with the amount of plays it took to complete the game.
can-play-card	This function evaluates a provided card can legally played ontop of another card according the standard UNO rules[6].
draw-cards	This function draws n amount of cards from the deck and inserts them into a given player's hand.

5.1.2 UNO-Agent.py.

The UNO Agent implementation is less focused on the game of UNO itself but more concerned with the most efficient way to play it. To create an agent in general, it is essential to create some performance metric that rewards the agent for moves that will get it closer to the goal state of winning, and that will punish it for the opposite. This is done through an evaluation function that tests if a certain game state will result in a win. If it does, it will return 1 as the performance metric, with -1 being the alternative. With that said, the following table depicts the functions from the UNO-Agent file and how they are utilized to build the agent and its evaluation function.

For this agent's evaluation function, a hand that has fewer cards is rewarded, up until the terminal state of no cards. Increases in the total amount of cards in the opponents' hands are also rewarded.

Due to the number of calls to this function by the Expectimax algorithm, it is important that its computational cost be kept low. The calculation for the reward based on the player's hand will run in $O(1)$ time. The calculation for the reward based on the opponents' hands will run in $O(n-1)$ for n total players. Therefore the function will run in $O(1) + O(n-1)$, or just $O(n)$ time. However, because n will usually be less than or equal to 10 during normal game play[6], the run time will be very close to constant time.

is-winner	This function returns if a given player is a winner based on a provided state.
uno-eval	This function servers as the evaluation function for the Expectimax UNO agent. It will determine the value of a state based on the previously defined criteria.
mini-max-uno-player	This function utilizes the aima-python repositories implementation of alpha-beta cuttoff search to run the Expectimax algorithm.
terminal-test	This function returns whether a provided state is a terminal game state as defined in a standard UNO game [6].

5.2 Testing

After finishing the implementation of the game and agent, the experimentation was able to begin. The experiments used in testing different aspects of the UNO agent were built in the experiment.py file, containing a variety of test functions. Those functions are as follows:

sim-game	This function runs a UNO game and updates experiment variables such as the number of wins or the round length based on the result of the game.
generate-win-percentages-per-game-size	This function runs a UNO game for a set number of iterations for a set of defined game sizes from a minimum to a maximum. These games are then used to calculate the win percentages per agent per each game size.
generate-overall-win-percentages	This function runs a UNO game for a set number of iterations for a set of defined game sizes from a minimum to a maximum. These games are then used to calculate the overall win percentage per agent regardless of game size.
generate-round-length-per-game-size	This function runs a UNO game for a set number of iterations for a set of defined game sizes from a minimum to a maximum. These games are then used to calculate the average game length per game size.
generate-overall-avg-round-length	This function runs a UNO game for a set number of iterations for a set of defined game sizes from a minimum to a maximum. These games are then used to calculate the average overall game length regardless of game size.
generate-csv	This function generates a csv file with a provided name. This function is used to generate csv data from the previously names functions for analysis.

Through the use of these functions,a file of data can be obtained containing the win percentages given certain situations. Some of the important situations that were identified before creating these specific functions included the win percentages of the UNO agent playing against a number of random agents, as well as in the case of multiple random agents playing each other as a control. Additionally, it was interesting to not only focus on performance in terms of win percentage but also on the efficiency of the agent, resulting in the logging of both the average round length and how it changes depending on the game size. This statistic however was not measured by the function in terms of time taken to play but instead the average number of moves per round. These results allowed for the analysis of the performance of the agent and algorithm and will be further interpreted and analyzed in the next section.

The control for the experiment consists of games consisting of agents that only make random legal moves. The experimental tests consist of the Expectimax agent playing against the same style of agents in the control. The experimental tests will always start with the Expectimax agent followed by n-1 random agents, where n is the total number of players for a game.

Each test will run 1000 games for each game size, with the game size ranging from 2 to 6 players. The average win percentage per game size per agent is calculated by dividing the agents' total wins for all iterations of games for a game size and dividing by the number of iterations. The overall win percentage per agent type, regardless of game size, is calculated by averaging the win percentages per game size for each agent type.

The average length of a game is calculated in a similar manner. For each round size, all game lengths (total number of moves from the initial to the final game state) are added and then divided by the number of iterations. The overall average game length, regardless of game size, is calculated by averaging the average game lengths per game size.

6 Results

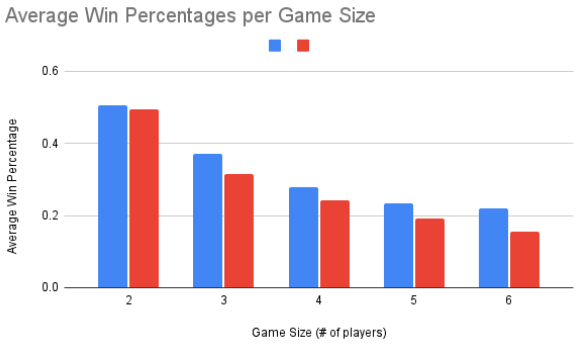
6.1 Statistics

The main results that we obtained through the experimentation phase were separated into two main categories: the overall win percentages for both a control group as well as the agent, and the average number of moves per round. As mentioned above, these statistics are averages from compiling the data of 1000 games and were obtained through testing on multiple different game sizes and are represented in the graphs below.

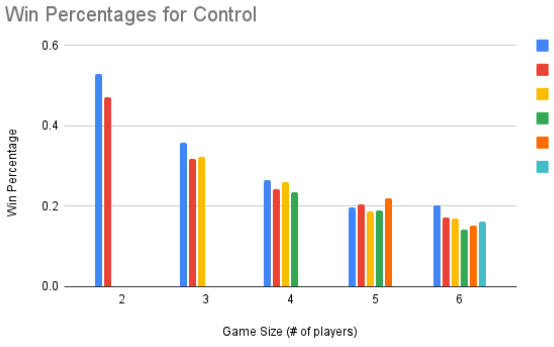
The first graph below represents the average win percentage of both the agent as well as a random agent for a game of size 2-6 players. The blue columns represent the performance of the agent using the Expectimax algorithm while the red represents an average of the performance of the arbitrary agents that the Expectimax agent is playing against. The red columns were obtained by averaging the win percentages of every random agent player. For example, in a game with six players, the Expectimax agent has a win percentage of 22%, and the other 5 random agents together have an average of 16%. With this said, the percentages of all agents individually

do indeed add to 1 for each game size, although that is not necessarily represented in the graph.

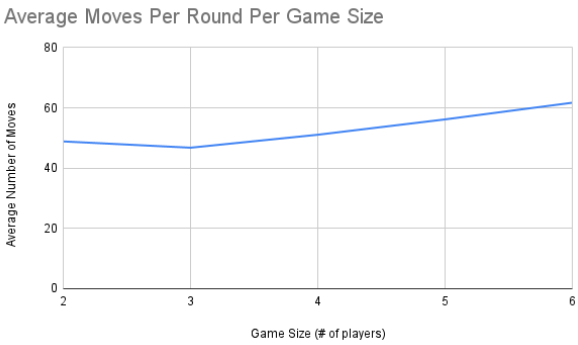
Another statistic that becomes clear not only through this graph but also through the generate-overall-win-percentages function in the experiment file, is the average win percentage of the different agents regardless of game size. Although grouping the data by the number of players definitely portrays a more accurate picture of the win percentage, it is also important to compare overall statistics. With that said, the overall average win percentage for the Expectimax agent is 32% whereas it is only 28% for the random agent.



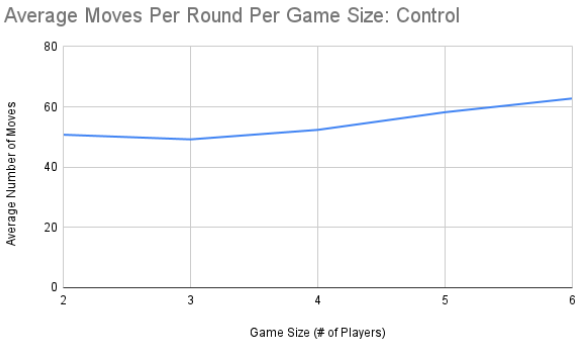
In addition to the data that we collected above involving the agent against a random agent, it was important to obtain control data from running random agents against one another. The graph below illustrates the win percentages of each of the random agents when they are playing together. Each game size has the same number of columns corresponding to the number of players, and all percentages add to 1.



The last category in which data was collected from the agent was involving the average number of moves per round for both the Expectimax agent as well as the control. The first graph represents the number of players vs the corresponding number of moves per game when playing multiple rounds of UNO with an Expectimax agent.



We also monitored the average number of moves per round in the situation of multiple random agents playing one another, which is detailed in the second graph. When referring to the average number of moves per round, it is specifically referencing the number of moves from the initial to the final state of the game, with regards to the game path and tree. With that said, on average, these statistics were 1-3 moves greater than those in the first graph, however, a significant effect was not made.



6.2 Analysis

All of the results above yield interesting conclusions. As can be seen in the first graph, the UNO agent does on average perform better than an arbitrary agent. However, although it does perform better as expected, it doesn't result in that large of a difference. When playing against only one player, it has a 1% lead over the random agent, although it does grow to be a 6% increase when the game size is 6. While it is still important to note this increase, it is also important to keep in mind how this could be affected by outside influence, which will be touched on later on. Another observation from this graph has to do with how the win performance does increase as more players join the game. This demonstrates how the Expectimax agent thrives in more complex game situations, which really illustrates how it does have at least a slight upper hand when it comes to win percentages and overall performance.

However, when comparing the graph of the agent with that of the control, it becomes clear that this small increase in performance might not be only a result of the Expectimax agent. In the second graph, the average win percentage for each control agent is visible, demonstrating that the performance of random agents does tend to vary by quite a lot. It illustrates that the win percentages of random agents are quite unreliable and could be the cause of why it appears as if the Expectimax agent is performing better than random. With that said, it is important to take this graph into account as a control when analyzing the first figure and in understanding the behavior of both the game and random agents.

With all of this said, it is apparent that there could be bias within the results. When collecting said data, the agents always started in the same order, with the Expectimax agent specifically always starting first. In the control, it can be seen that there seems to be a trend where the first random agent tends to outperform the other agents. With that being the case within the control group, it is also possible that is where the increase in performance in the Expectimax agent originates from. When mentioning the first agent, it refers to the agent who plays the first card in gameplay. Since that allows the first agent to play a card before any others, it may be a large factor in why it tends to have the highest win percentage. This is due to UNO being a game that is won due to being the first player to run out of cards. Having this control graph as a result is extremely necessary in order to see a point of vulnerability in the results, and while it does not completely invalidate previously stated conclusions, it is still important to note when looking at this figure.

Finally, another aspect of the results that was interesting to receive was the average number of moves per round. Even with the number of players increasing, the number of moves per game wasn't severely affected, although it did increase with the game size more consistently. This becomes apparent in the table through the average number of moves for two players being approximately 49 whereas the average for six players was closer to 62. This being the case makes sense in the context of the game because if there are more players, there is likely to be a greater number of cards being dealt and therefore played in any given round. However, this is not the only important difference to note. When compared with the average number of moves per game in a game situation with only control players, the agent does slightly reduce these numbers. This is likely due to the Expectimax agent being slightly more efficient in choosing what the best possible move is, instead of just a random one. While this statistic does not necessarily result in solving the problem we set out to when creating the agent, it is still an interesting one to explore, as it gives a bit of inside information on how the Expectimax algorithm is contributing to the agent's efficiency.

From this analysis, it becomes apparent that the implementation of this agent in particular did not significantly impact the win percentage when playing UNO, which was the goal. With that said there are a few shortcomings to consider. A few potential causes of these results could be the incorrect implementation of the agent itself, the use of an unproductive game theory algorithm for the type of game, or simply the problem itself. First off, it is important to give thought to the situation in which we implemented the agent. The testing was implemented by having only random agents playing against the agent, without doing any research into how accurate the random agent is within gameplay. The aima-python code repository was also utilized to implement certain features of the agent, which could be acting in ways that were unexpected upon analyzing certain features. Lastly, it is possible that the limitation of the game tree depth could have played a role in the accuracy and therefore the agent's ability to make the right actions in order to win. The limitation of this depth was originally applied due to efficiency in the time taken to make decisions and allow for multiple test cases, however, it could have ended up making a negative impact on the results. These three things as well as many other small mistakes could have resulted in the incorrect implementation of the agent in tandem with the Expectimax algorithm and therefore could have skewed the results. However, it is possible that had we implemented everything correctly, the Expectimax algorithm still wouldn't have been ideal. Although we did our research on different types of algorithms out there, it is difficult to know for sure how each algorithm will act when paired up with an agent to play a game like UNO. Having chosen this algorithm and rolled with it, could have resulted in minimal performance if it is not serving the role that it needs to within the evaluation function. Lastly, another major cause of shortcomings could be due to the misunderstanding of the game and the problem. Having chosen UNO as the game for the agent, it was impossible not to note the role that chance came into multiple different aspects of this game. However, I think this characteristic of the game was underestimated and could result in any agent having a relatively difficult chance of increasing its win percentage beyond a certain point.

7 Conclusion

Throughout the course of this report, the focus has been on successfully creating an AI agent that can win games of UNO with a higher probability than random chance. After doing significant research on what game theory algorithms best worked with this game in particular and creating a plan for how to approach the problem, it became clear that an Expectimax algorithm would best fit the problem. After the approach was identified, it was possible to implement not only the agent but also the game itself, along with a way to run the experiments needed to test the performance of the agent. After collecting the data, it was concluded that the agent did increase both the win performance and

efficiency from that of the random agent, however by an amount not significant enough to make a large difference. Therefore, these conclusions were analyzed to be relatively insignificant due to some of the bias in testing and overall shortcomings that were discovered after receiving the results and comparing them with the control.

With that said, although some progress was made from the creation of the agent, there is still a lot of progress out there to be made. However, before making changes to the implementation of the agent itself, it would be helpful to reevaluate the current condition of the agent and the results that it yields. Due to the bias introduced as a result of running the agents in the same order every trial, it is important to later test the agent in different ways to verify the results given. After that is completed, progress in solving the problem can be made in a few different areas. The most prominent of these areas of progress being trying out different game theory algorithms. Although at first impression it seemed like Expectimax was the most fitting algorithm for a UNO agent, another algorithm might allow for better performance. These algorithms are definitely a cause for more research and work in the future if the goal is to maximize performance even more.

As shown in Section 3, reinforcement learning and deep learning have already been utilized for creating a UNO agent and other agents for stochastic games. Perhaps in order to make up for the pitfalls of the Expectimax algorithm in large state spaces, a trained model can be used as an evaluation function. A model could also be trained to eliminate states from the belief states in order to shrink the state space. This is an area for potential research.

8 Contributions

The following table depicts the contributions of both group members in this project. It refers to the sections in which each group member wrote, however the code was implemented by Andrew Guerra and the graphs were made by Grace Condie.

Grace Condie	1, 2.2, 2.3, 3.1, 3.2.1, 3.2.2, 4.1, 4.2, 5, 5.1, 5.1.1, 5.1.2, 5.2, 6.1, 6.2, 7
Andrew Guerra	2.1, 3.2.1, 3.2.3, 3.2.4, 3.3, 4.1, 5.2, 7

References

[1] Olivia Brown, Diego Jasson, and Ankush Swarnakar. 2020. Winning Uno With Reinforcement Learning. (Nov. 2020). https://web.stanford.edu/class/aa228/reports/2020/final79.pdf

[2] Jd Dan Davenport. 2016. Minimax AI within Non-Deterministic Environments. (Aug. 2016). https://doi.org/10.1145/1188913.1188915

[3] Jose M. Font, Tobias Mahlmann, Daniel Manrique, and Julian Togelius. 2013. A Card Game Description Language. 254–263. https://doi.org/10.1007/978-3-642-37192-9_26

[4] Hajime Fujita and Shin Ishii. 2005. Model-based reinforcement learning for a multi-player card game with partial observability. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. IEEE, 467–470.

[5] Raluca Gaina, Martin Balla, Alexander Dockhorn, and Raul Montoliu Colás. 2020. TAG: A tabletop games framework. CEUR Workshop Proceedings.

[6] Mattel Inc. 2001. Uno Instructional Sheet. (2001).

[7] Moyuru Kurita and Kunihito Hoki. 2021. Method for Constructing Artificial Intelligence Player With Abstractions to Markov Decision Processes in Multiplayer Game of Mahjong. *IEEE Transactions on Games* 13, 1 (2021), 99–110. https://doi.org/10.1109/TG.2020.3036471

[8] Elizabeth Lipin, Brian Tang, and Stephenie Worthy. 2023. Durak, Don't Be the Fool: A Monte Carlo Tree Search Approach. (Jan. 2023). https://www.bjaytang.com/pdfs/EECS_592_Project.pdf

[9] Frans Oliehoek, Matthijs TJ Spaan, Nikos Vlassis, et al. 2005. Best-response play in partially observable card games. In *Proceedings of the 14th annual machine learning conference of Belgium and the Netherlands*. 45–50.

[10] Shayegan Omidshafiei, Karl Tuyls, and Wojciech M. Czarnecki et al. 2020. Navigating the landscape of multiplayer games. Nature Communications. https://doi.org/10.1038/s41467-020-19244-4

[11] Bernhard Pfann. 2021. Tackling the UNO Card Game with Reinforcement Learning. (Jan. 2021). https://towardsdatascience.com/tackling-uno-card-game-with-reinforcement-learning-fad2fc19355c

[12] Stuart Russell and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. Vol. 4. Pearson Education, US.

[13] Nathan Sturtevant. 2003. Multi-Player Games: Algorithms and Approaches. https://www.proquest.com/docview/305351356?pq-origsite=gscholar&fromopenview=true

[14] Ahmad Zaky. 2014. Minimax and Expectimax Algorithm to Solve 2048. (May 2014). https://osf.io/az7vf/download

[15] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. 2020. RLCard: A Toolkit for Reinforcement Learning in Card Games. (Feb. 2020). https://arxiv.org/pdf/1910.04376.pdf