Monte Carlo in Monte Carlo: Pathway to Victory in UNO

Wenyang Hui, Mengyun Liu, Chibin Zhang

December 31, 2020

Abstract

The game UNO, whose name originated from the Latin "One," has simple rules that could be quickly mastered by a three-year-old. But make no mistake, deeply concealed in the simplicity of rules is the combinatoric complexity of game states. The easy to learn but hard to master characteristic of this game makes it an exciting target for artificial intelligence research. In this paper, we create novel adaptations to Monte Carlo Tree Search, a popular game search technique mostly applied in board games, and tailor it to UNO's highly stochastic nature and complex state space. To reflect on what we have learned in class, we also apply strategies such as informed search, expectimax search, and reinforcement learning in our greedy agent, expectimax agent, and deep Q-learning (DQN) agent. We evaluate the effectiveness of our agents by letting them compete against random agents and among themselves. Our experimental result suggests that our Monte-Carlo Tree Search - Dynamic Bayesian Model (MCTS-DBM) agent surpasses the random agent significantly and beats state-of-the-art UNO agent from [Olivia et al., 2020] by a wide margin.

1 Introduction

The game UNO comes in 2 flavors: official or common. Often distributed with a box of UNO cards is a folded sheet of rules of UNO as in [Mattel games, 2020]. The official rulebook is point-oriented, meaning that for a player to win, he would have to score points and the first person to score 500 points wins the game. Keeping track of points while enjoying the game is indeed a cumbersome task. In reality, when folks play UNO, they seldom refer to the rulebook and instead play by the assumed common rules described in [Wikipedia contributors, 2020]. Common-flavored UNO game plays much similar to a shedding-type card game, i.e., the first player that discards all his hand cards is the winner.

Besides drawing or playing a card, some rules require players to act upon certain events, e.g., calling out "UNO" when playing the next-to-last-card or challenge a player when he plays the "Wild +4" card. Such rules are added for variety and interactivity, but modeling them in a program complicates implementation and analyses. For the sake of clarity, we do not consider them as a valid action for our agent.

1.1 Deck

A regular UNO Deck consists of 108 cards. Judging by the color, a card could be red, yellow, green, blue, or "Wild", in this case, often colored black. Judging from how they influence gameplay, the cards can be either numbered or functional. Numbered cards, when played, is only discarded. Functional cards, on the other hand, have special purposes like forcing the opponent to draw a card that can be used to the user's advantage. A complete listing and count of each card is given in the following table1.

1.2 Game Mechanics

Given the vast array of actions, sources of uncertainty, and our project's limited timespan, we decided to simplify UNO's mechanics. To do so, we made several simplifying assumptions. The exact game rule is shown in the following.

- Round start
  Flip the card on top of the draw deck. If a non-numbered card appears, put it into the discard pile. Repeat until a numbered card appears. Start the game clockwise with the dealer.

Type	Count
Number 0	2
Number 1 - 9	4
Reverse	4
Skip	4
Draw 2	4
Wild	4
Wild +4	4

Table 1: Types of card in UNO and their respective count

- Play
  A player could choose to play or draw a card. When he plays a card, his choice must have the same color or the same number as the previous one, or it could be a wild card. A "Reverse" card reverses the direction of the game, e.g., clockwise to counterclockwise. A "Skip" card skips the next opponent's turn. A "+2" card adds 2 cards from the deck to the next opponent's hand and skips his turn. A "Wild" card is wild in the sense that it can succeed any cards. A "Wild +4" card is similar to the prior, except that it adds 4 cards to the next opponent and skips his turn. If a player has no card to play, then he must keep drawing from the deck until a valid successor appears, or he is satisfied, whichever comes first.
- Winning Condition
  The first player to discard all his cards is declared victorious.

1.3 Difficulty

UNO is a typical stochastic imperfect information game. As the fact that there are few works about UNO suggests, the highly stochastic nature leads to the difficulty in implement an intelligent agent. In brief, the difficulties of implementing a UNO agent is list as follows:

- Irregular game tree caused by "reverse cards".
- Huge state space.
- Many hidden states.
- Sparse reward.
- Highly stochastic environment.

1.4 Contribution

In this paper, we explore game search techniques taught in class in a real-world setting and dig into the nitty-gritty details of implementing an agent for games with vast imperfect information and huge state space. To the best of our knowledge, we're the first to use tree search method including Expectimax and Monte-Carlo Tree Search together with Dynamic Bayesian Model to solve the card game UNO. Finally, Our experimental results complement class knowledge by providing a side by side, quantitative, and thorough evaluation of different agents.

2 Related Work

[Mnih et al., 2015] showed how deep reinforcement learning help agents excel in retro video games on the Atari platform. Their evaluation reports the DQN agent performing at a super-human level in 29 out of 49 games, a promising result that largely contributed to the proliferation of reinforcement learning both inside

and outside academia. Their monumental achievements greatly motivated and spurred the authors to apply DQN to game problems.

[Coulom, 2006] described the application of the Monte Carlo method to game-tree search and coined the name Monte Carlo tree search, which later gained tremendous popularity in the artificial intelligence research community. He demonstrated his method's effectiveness on a once deemed impossible AI puzzle, go, achieving dan master on a constrained 9 by 9 go board. This same technique was later adopted by Deepmind, conceiving the nowadays household name AlphaGo. While Go is a board game and UNO is a card game, a striking similarity that they both share is the simplicity of rules and the complexity of states. We believe employing MCTS to the UNO scene will be beneficial.

Games and reinforcement learning have long been a research genera in artificial intelligence; nevertheless, few have investigated the effectiveness and practicability of such an approach in UNO until very recently. [Zha et al., 2019] designed a collective reinforcement learning framework for common card games, called RL-Cards. We appreciate their work. However, their game interfaces are complicated and inefficient. As a result, we referenced their implementation and tailor the game environment to our agents' needs. [Olivia et al., 2020] released their work on a UNO DQN agent. We cite experimental results from their paper and use them as a metric for testing our method's effectiveness.

3 Methodology

3.1 Open Loop Tree Search

Open loop is an idea proposed by [Perez Liebana et al., 2015]. In a close loop game tree search, the game tree holds all the information and simulate the game itself. In determistic games, an agent can perfectly simulate the game, and once a node of state s is added to the tree by doing action a from state s', the agent does not need to worry about it can still redo that process at s'. However in stochastic imperfect information games the agent cannot make such guarantee. Some proposed chance node and expectations to model this uncertainty, but the existance of massive opponent nodes and chance nodes make the game tree extremely large and inefficient to sample. Moreover hidden information exists in this tree, which close loop tree agent cannot handle naturally. On contrary to close loop game tree search, open loop only records the information about the current agent and excludes the other information from the game tree. In open loop game trees, only actions are considered in tree nodes while the others part is done by a simulator. Resembling Q-learning, this method does not care about the complicated and sometimes intractable transitions in the environment, which is useful in simulation based tree search such as MCTS in stochastic imperfect information games.

3.2 MCTS-DBM Agent

To *explore*, or to *exploit*, that is the question. As the computational resource is ultimately bounded, a common trade-off that tree search algorithms face is choosing to go deeper or wider. Going deeper means that we could arrive at the terminal state, have full certainty of the outcome, but at the cost of landing on a suboptimal option. Going in breadth let us explore comprehensively, in exchange for limited depth. We don't miss out on the optimal option if there are any, but we would have approximate utilities for the states we've seen so far. Traditional game search strategy like Minimax search requires exploring the shallower layers before proceeding to the deeper layers, even when used together with alpha-beta pruning, resulting in poor performance in real-world problems. On the other hand, MCTS quantitatively unifies exploration and exploitation through the Upper Confidence Bound formula (1), and strike a balance between the two via iterative expansion and simulation. In the following subsections, we explain how MCTS works and the adaptations we made to work on UNO.

3.2.1 MCTS

In essence, MCTS can be divided into 4 distinctive phases: selection, expansion, simulation, and back-propagation. In the selection phase, the MCTS algorithm will recursively select the child nodes of the current node that evaluates the highest UCB value from root. The UCB formula is given in equation (1), where x_i is the expected utility for state i, C is a constant controlling exploration, t is the total number of

simulations, and n_i is the number of times state i was visited. Due to the frame and scope of this paper, we do not offer a complete derivation of this formula but instead, give an intuition. A closer inspection, the UCB formula resembles the exploration function of Q-learning where the expected value of a state is added to some constant times the inverse of the times the state is visited. When n_i is small, the uncertainty of a given state is large, resulting in a large S_i value, making it more likely to be selected by the agent. When x_i is large, this means that the node's expected utility is high; exploiting the action that transits into this state will likely give good results, so it is also scored with high value. The exact choice as to which state is to be selected made by the agent is influenced by both of the two terms. In the expansion phase, children of the selected node are added to the tree, broadening the fringe. In the simulation phase, the MCTS algorithm goes deep down the tree sampling value from actual terminal states. The result of the simulation is then propagated back up to the root to improve the accuracy of approximation of the intermediate node's value. After running enough iterations of the aforementioned 4 phases, we gain enough confidence in the optimality of certain actions and take.

$$S_i = x_i + C\sqrt{\frac{\ln t}{n_i}} \tag{1}$$

Although we believe that MCTS is the right direction to take when tackling UNO, some issues prevent us from using vanilla MCTS. One issue is intractable transitions. This can be solved by the open loop method mentioned above. Another issue that we stumble across is that it requires the hidden information, which our agents do not have direct access to, to start simulation. Assume that the prior distribution of each players' cards follows from the uniform, as time goes on, whenever a player plays a card based on their observations. Then the process of players playing cards can be seen as a Dynamic Bayesian Model(DBM). Similar to HMM, DBM is also a known method for analyzing temporal data. To begin simulation, we can either directly use an oracle to mimic the other agents playing or predict their hand and then infer their action. Because exact inference is computationally intensive, we use approximation algorithms for prediction.

3.2.2 Hand prediction using DBM

In prediction we draw inspiration from Hidden Markov Models (HMM), and particle filtering algorithm learned from class to overcome the huge and complex transition and emission. As a result we proposal a Monte Carlo based approach to approximately predict the distribution of the hidden states. The algorithm is shown in Algorithm 1, and the soundness is proved in Theorem 3.1.

Algorithm 1: Monte Carlo based hidden state prediction

```
After the main game has started, randomly initialize a number of subgames;
while The main game is not terminated do
    After one agent has done an action in main game, require the same agent in each subgame to
     perform the same action as in the main game;
    if The aciton can be performed successfully then
        Set weight of this subgame to 1;
    else
        Randomly perform a valid action and set weight of this game w_a, where 0 < w_a < 1;
        Here, different from pure particle filtering, we do not set the weight to the
         hard value 0 to prevent overfitting.
    end
    Resample the subgames based on the weights
end
```

Here after each time a player plays a card, we check if the subgames agrees with our the observation and assign higher weights to particles that do and lower ones that don't. Passage of time causes the particles to gather around the highly probable cards, and our opponent model becomes more accurate. We select the most probable cards as input for MCTS queries. In other words, we *determinize* the most likely explanation as a point estimate to deal with UNO's uncertainties.

Theorem 3.1 (Soundness of MC approximation)**.** *The proportion of correct predictions on hidden states does not decrease after a sequential of updates. i.e.*

$$P_{t+1}(h_{gt,t+1} \mid a_{1:t}) \geq P_t(h_{gt,t} \mid a_{1:t-1}),$$

where $h_{gt,t}$ is true hand at time t.

Proof. Denote the weight function of emission w, the performed action a. Then we have

$$w(h_{gt,t}, a) = 1 \tag{2}$$

$$P_{t+1}(h_{gt,t+1} \mid a_{1:t}) = \frac{w(h_{gt,t}, a_t) P_t(h_{gt} \mid a_{1:t-1})}{\sum_{h_t} w(h, a_t) P_t(h_t \mid a_{1:t-1})} \tag{3}$$

Since $w(h, a_t) \in (0, 1]$, $0 < \sum_{h_t} w(h, a_t) P_t(h_t \mid a_{1:t-1}) \leq 1$. Therefore

$$P_{t+1}(h_{gt,t+1} \mid a_{1:t}) \geq w(h_{gt,t}, a_t) P_t(h_{gt} \mid a_{1:t-1}) = P_t(h_{gt} \mid a_{1:t-1})$$

□

Then we can carry out game simulation based on the sound prediction on the hidden states. Here we combine Random Forest and MCTS to reduce the variance of value of the game tree, where each individual tree is a sample uniformly drawn from the subgames.

3.3 Expectimax Agent

Expectimax search, as a traditional game tree search algorithm, is based on close loop game tree. However it is not feasible in this UNO, since even if the hidden states are not considered, the tree size is too large to perform a depth 1 expectimax search. The main cause is that a number of chance nodes and opponent nodes, and their large branching factor lead the game tree to be oversized. We exploit the idea from open loop. Although open loop method is used in simulation based methods, we can take a similar approach that aggregate all other nodes except for the nodes of the expectimax agent. And in the transitions, we make simple but reasonable assumptions that the unseen cards uniformly compose the opponents' hands and the deck, and the opponents randomly play a valid card according to the previous card. Then the process that the opponents playing cards can be viewed a Markov Chain. Since we only care about the outcomes of opponents playing, then all the chance nodes and opponent nodes can be aggregated into one single chance node. The expectimax only consider then expectation of the "previous card" we the expectimax agent face when it is in the next turn it plays. Then the exponential explosion is reduced. The detailed calculation of the consequencies of opponents playing cards is described in Algorithm 2.

Algorithm 2: Calculating the probability of each "previous card"

$U \leftarrow$ the vector of unseen cards of a certain kind.;
$m \leftarrow$ the number of all unseen cards;
$P_a \leftarrow$ be the indicator vector of which action expectimax agent take.;
Save number of cards in the opponents' hands in l;
for *n in l* **do**
  $P_h = 1 - (1 - (U/m))^n$;
  P_h `is vector of the probability whether each card is in hand`;
  $P_j \leftarrow$ the indictor vector that the ith entry is 1 iff i can be played after j;
  $T[j] = P_h * P_j$ Elementwise product;
  $P_a = T * P_a$ Matrix production;
end
Return P_a

Algorithm 2 gives all possible outcomes after the current agent playing action a. With these probabilities, our agent can maximize the average of all possible consequences, which is the same as vanilla expectimax.

3.4 Other Agents

3.4.1 DQN Agent

Recall what we have learned about reinforcement learning in class. To solve a model-free MDP problem, we would have to specify the Transition function $T(s, a, s')$ and the Reward function $R(s, a, s)$. Such an approach worked well in limited and constrained setting the like *Gridworld* where the number states never exceeded a hundred and viable actions at each state are within 4. However, the vanilla Q-learning algorithm does not scale to real-world problems. It immediately falls apart in the setting of UNO where states explode to the magnitude of game duration. Precise representation of state and action pairs would exhaust memory very quickly, even in today's supercomputers, making it intractable both in time and space.

A popular technique used in combination with reinforcement learning to cope with the state explosion problem is to use a deep neural network as function approximators for q-values. An intuition for such an approach is that while the states are indeed unique, most of them only differ by only small variation. A neural network will hopefully capture the similarity between states, generalize, and densely represent experience compared to pure enumeration. In a nutshell, we trade loss of accuracy for computational efficiency.

We adopt such a technique in our DQN agent. A subtlety that arises from providing inputs to the neural network is that neural networks expect fixed-length input while the number of cards in an agent's hand can vary. We solve this problem by converting the list of cards into an array of size $54 \times 5 = 270$. Since there are 54 kinds of cards in UNO, then this array is a binary array where value is 1 in $i + (54j)$ if it has j card i. Our neural network layer is a 3 network with 2 hidden layers with 512 channels, 270 input channels and 60 output channels. An array of size 270 is fed to the network to obtain the approximated q-value.

3.4.2 Reflex Agent

On a high-level, the reflex agent's strategy is to mimic what a human player would do: greedily minimize the number of cards in hand. It tends to play a card rather than draw a card. In fact, it only chooses to draw as a last resort. When there are multiple cards in hand, it prefers playing numbered cards against functional cards. The fewer cards it has in hand, the less probable it is going to have a card match in color or number with the previous card; saving a "Wild" card appears to be a good strategy. Implementation wise, valid actions are sorted in order of numbered cards, utility cards, and draw. The greedy agent always chooses the first viable action.

4 Experiment

In this part we elaborate the experiments about our agents.

4.0.1 Experiment setting

We listed how the parameters of out agents are set in this part.

1. MCTS-DBM Agent: $c : 5$, trees in random forest: 5, number of simulations for each tree: 100, agent used for simulation: reflex agent.

2. Expectimax Agent: depth = 2, evaluation function:

$$\text{number of color in hand} - 2 \times \text{number of cards in hand} + \text{number of wild cards in hand}$$

3. DQN Agent: Size of experience replay buffer: 10000, moves between trains: 2, train samples used for one iteration: 32, learning rate: 1e-4, discounting factor: 1, total training iterations: 25000, ϵ greedy with $\epsilon = 0.1$.

4.1 Training statistics about DQN

We trained our DQN agent for 25000 iterations and converaged, and average loss over time is depicted in Figure 1 below. We visually observe that the loss curve flattens out after about 10000 iterations and converages.

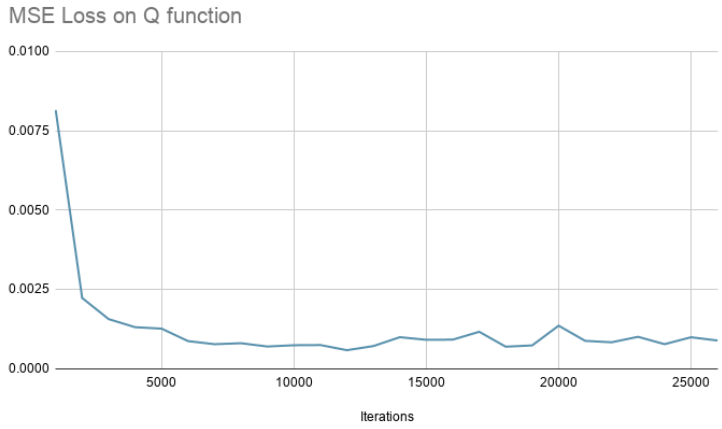

Figure 1: The train loss on Q function against iteration

4.2 Wining rate evaluation

We evaluate the effectiveness of the 4 agents by performing 2 types of tournaments. In the first tournament, the player count is set to 4, and the selected agent will play against 3 random agents. We record and the number of times each agent wins and divide by the total number of rounds played to get the win rate. In the second tournament, we focus on the pairwise relative strength of agents and let them play in 1 vs. 1 matches.

4.3 Against Random Agents

As shown in table 2, all agents outperform the random agent, but with a varying degree of significance. Comming in the bottom of the list is the MCTS-DBM agent with a win rate of 0.96. We initially expected the expectimax agent to be at least better than the reflex agent because it considers the future. We attribute its marginal improvement to its limited tree traversal depth and high uncertainty of the game. Once again to our surprise, the reflex agent is almost on par with the DQN agent. Our prior anticipation was that the DQN agent would be the second, if not the best, as we put lots of thought process and validation into it, and it is practiced by other researchers. As mentioned in the previous section, the reflex agent implements a policy close to how human players play the game. Such heuristic, though very simple, deemed very useful. The MCTS-DBM agent obtains the best performancce, but it is also the most computationally intensive one. [Olivia et al., 2020] is another work about exploiting DQN in UNO, Their evaluation metric is the same as us. Although they do not publicize their code so we cannot compare it with us in code level, their problem setting should be the same us ours according to their description. From the win rates, ours agents outperform them by a wide margin.

Number of random agents	1	2	3
[Olivia et al., 2020]	/	0.64	0.46
Reflex	0.95	0.91	0.89
DQN	0.96	0.93	0.91
Expectimax	0.96	0.94	0.92
MCTS-DBM	**0.96**	**0.95**	**0.94**

Table 2: Win rate against random agents (1000 rounds)

	MCTS-DBM	DQN	Reflex	Expectimax	Random
MCTS-DBM	/	0.55	0.60	0.47	0.98
DQN	0.45	/	0.56	0.48	0.96
Reflex	0.40	0.44	/	0.47	0.95
Expectimax	0.53	0.52	0.53	/	0.96
Random	0.02	0.04	0.05	0.04	/

Table 3: Win rate in 1 vs. 1 matches (100 rounds)

4.4 Against one another

While competing against random agents give us a ranking of the agents, in real life, we seldom judge a player's strength by how well he played against a jerk. Thus we put the agents 1 vs. 1 matches to gain a clearer picture. The results3 is in line with table 2. Relative strength against random agents implies having the upper hand in an 1 vs 1 match.

In this table, Expectimax agent, to our surprise, beats MCTS-DBM agent. A potiential reason for this is that MCTS-DBM uses a reflex agents to simulate a game. MCTS-DBM agent should beat expectimax agent if it uses expectimax to simulate.

5 Conclusion

In this paper, we make the following contributions. We provide a comprehensive comparison of game playing techniques taught in class, put in real-world settings. We novelly adapt MCTS, which is usually employed in deterministic and fully observable games, to UNO, a game with complex state space and largely imperfect information viewed from the agent's perspective. We believe our determinization technique is generic and practical enough to serve as a general framework to think about card games and real-world problems where partially observable states are the norm.

5.1 Future Work

Simuation in MCTS is slow, optimization in game logic can improve the efficiency in simulation. Domain specific knowledges should also be exploited. One possible solution is state isomorphism. In alphaGo Zero [Silver et al., 2017], the board is rotated and flipped to enlarge the train set, since the rotated and flipped boards contains exactly the same information as the original one. This method can be used in UNO as well. If all the isomorphism states can be mapped to the same state, the state space will be decreased exponentially and sample efficiency will increase also exponentially. However it is hard to design such a function when all factors are considered.

A External Code

- Numpy for vectorized calculation.
- Pytorch for DQN.
- Pygame Zero for GUI.
- The GUI implementation is modified from `https://github.com/bennuttall/uno`

References

[Coulom, 2006] Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer.

[Mattel games, 2020] Mattel games (2020). Uno instruction sheet. [Online; accessed 31-December-2020].

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[Olivia et al., 2020] Olivia, B., Diego, J., and Ankush, S. (2020). Winning uno with reinforcement learning. https:/web.stanford.edu/class/aa228/reports/2020/final79.pdf.

[Perez Liebana et al., 2015] Perez Liebana, D., Dieskau, J., Hunermund, M., Mostaghim, S., and Lucas, S. (2015). Open loop search for general video game playing. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 337–344.

[Silver et al., 2017] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *nature*, 550(7676):354–359.

[Wikipedia contributors, 2020] Wikipedia contributors (2020). Balance puzzle — Wikipedia, the free encyclopedia. [Online; accessed 31-December-2020].

[Zha et al., 2019] Zha, D., Lai, K.-H., Cao, Y., Huang, S., Wei, R., Guo, J., and Hu, X. (2019). Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*.