# Reinforcement Learning in Settlers of Catan

1st Max Gerber
*Department of Aerospace*
*University of Colorado Boulder*
Boulder, Colorado
max.gerber@colorado.edu

2nd Mitchell LaRocque
*Department of Computer Science*
*University of Colorado Boulder)*
Boulder, Colorado
mitchell.larocque@colrado.edu

3rd Maxwell Van Sickle
*Department of Computer Science*
*University of Colorado Boulder*
Boulder, Colorado
maxwell.vansickle@colorado.edu

***Abstract*—This paper investigates the efficacy of employing Monte Carlo Tree Search (MCTS) and Deep Q-Networks (DQN) algorithms in the context of playing a reduced version of the strategic board game Settlers of Catan. By integrating these advanced techniques, we aimed to enhance the decision-making capabilities of AI agents to achieve superior performance in gameplay against both heuristic and random players. Our results indicate that AI agents employing MCTS can outperform both heuristic and random players, showcasing their ability to explore the game state and understand the reward structure. Furthermore, leveraging DQN allowed us to train AI agents capable of learning optimal policies directly from raw game states. Our findings reveal that AI agents equipped with DQN achieved adequate success rates against random opponents, showcasing their ability to learn effective strategies through reinforcement learning. The insights gained from our experiments pave the way for further research into exploring the intricacies of the full version of the game.**

## I. Introduction

Settlers of Catan is a board game that centers around strategic building of settlements, cities, and roads. Players acquire resources through dice rolls, trading, and development cards, which they can use to build said settlements, cities, and roads. The board itself is composed of hexagonal tiles that represent various resource regions, and whose configuration can change with each game. An example of the beginner board which was used for testing can be seen in Fig. 1. The objective is to be the first to reach ten "Victory Points", which are earned by building structures, holding the longest road, the largest army, and purchasing development cards. The game emphasizes a dynamic mix of strategy, negotiation, and resource allocation, making it perfect for exploring complex decision making processes such as those found in reinforcement learning algorithms.

In order to formulate the state of the game so that it would be more compatible with reinforcement learning algorithms, we decided to reduce the state and action space by removing the ability to purchase and use development cards as well as by not including the robber. Doing this allowed us to employ the Monte Carlo Tree Search (MCTS) and Deep Q-Networks (DQN) algorithms. MCTS is designed to navigate through large decision spaces by sampling actions and evaluating their potential rewards. In an environment with deterministic transition probabilities such as those for Settlers of Catan, this makes MCTS a very effective algorithm. DQN combines the principles of deep neural networks with the classic Q-learning algorithm to achieve state-of-the-art performance in a wide range of tasks, including game playing, robotics, and autonomous systems. DQN leverages deep neural networks to approximate the Q-function, which estimates the expected cumulative reward for taking a specific action in a given state. By training DQN agents to learn optimal policies through interaction with the game environment, we aim to develop intelligent agents capable of making strategic decisions and outperforming traditional heuristic-based approaches. Through experimental evaluation and analysis, we demonstrate the effectiveness of DQN in learning to play Settlers of Catan and provide insights into its strengths and limitations in this domain.

## II. Background and Related Work

### A. Background

Monte Carlo Tree Search (MCTS) is solved with "a state $s \in S$ and a value function estimate $\hat{V}$ , it attempts to run the value function iteration for a fixed number of steps, say $H$, to evaluate $V^{(H)}(s)$ starting with $V^{(0)} = \hat{V}$" [4]. This methodology of using MCTS with a value function estimate has been employed in a variety of games ranging from chess to AlphaGo Zero [4]. Similarly Deep Q-Networks (DQN) have been very popular for learning how to play games, even surpassing the level of a human expert in three of seven games when tested by DeepMind [9]. The DQN algorithm for Atari "learned from nothing but the video input, the reward and terminal signals, and the set of possible actions–just as a human player would" [9].

### B. Related Work

There is ample evidence of prior Reinforcement Learning algorithms having a lot of success in board game environments [1]. Due to the popularity of Settlers of Catan, attempts to create a similar approach for making a Reinforcement learner already exist before our attempt. One of the more popular ones, CatanAI, attempts to make a framework built in Python with AI agents trained using Reinforcement Learning [8]. This version of Catan is interactive, as a full board is generated and displayed, and the user can click on the screen to complete actions. The user can then play against AI agents or against other local players playing on the same machine.

While this project had the same end goal as us, their implementation was quite different in terms of state representation, so it mainly used as a proof of concept. Especially since their representation and methodology was so complex and intricate, we decided to move in a different direction to represent the game of Catan.

Additionally, looking at other work from student groups at

## III. Problem Formulation

### A. Board/Game State

Another adaptation we found through our research of existing implementations was PyCatan2, a python module for running games of Settlers of Catan [6]. This implementation provided necessary basic implementations of a game, including Game State (who has what resources and what buildings on what tiles), Resources for a given roll, Board Visualization, and automatic methods for determining valid places to build a settlement/city/road & valid trades a player can do (4:1 and 3:1 or 2:1 with harbor).

This board adaption was much simpler than the previously mentioned CatanAI, and thus was used as the basis for the game state. By using this prebuilt board, we could focus our time on creating methods that would play Settlers of Catan automatically, rather than spending it on creating a board implementation. Additionally, PyCatan offered variations between a "Beginner Board" and "Random Board" that would allow us to test our implementations on brand new boards. As shown in Fig. 1, this is a beginning instance of a "Beginner Board" before players place their initial settlement.

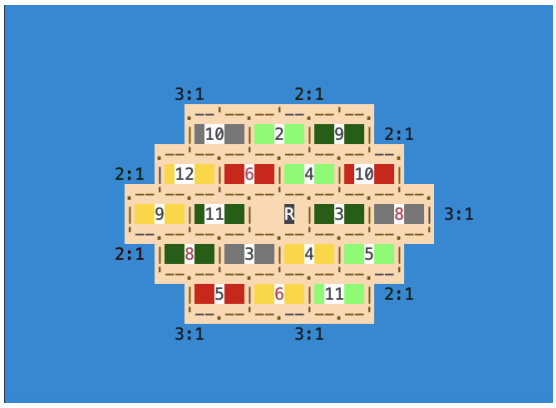


Fig. 1. Beginner Board Setup.

When players place their first settlement, they can place it an any valid intersection between tiles, as long as it is more than one intersection away from the next settlement. The resource type of the adjoining hex tiles of the placed settlement determine which resources can be received from rolls, while the values on the tiles represent which number has to be rolled to get said resource. While this is just the most important rules for starting the game, a full list of rules is available at the Catan website [2].

With this board implementation, we were able to create a Game state with the following attributes:

- Board Class:
  - Hexes, defined by the resource type and six coordinates that form the hex
  - Intersections, defined by a single coordinate where a settlement or city can be placed.
  - Paths, defined by two intersections they connect where a road can be placed
  - Harbors, defined by the intersection they are attached to
- Player Classes:
  - Resources, how many of each five resources the player owns
  - Settlements, and which coordinates they are located at
  - Cities, and which coordinates they are located at
  - Roads, and which paths they are located on
  - Victory Points, how many points the player has, from 1-10

This was the game state that was passed to our methods and was updated with every turn.

### B. Game Actions

For the game actions, during any given turn, a player must take a minimum of one action from the following.

- Build City
- Build Settlement
- Build Road
- Trade ____ for ____
- Next Player's Turn

These actions however are restricted by any player's available resources at a given turn.

### C. Transitions

Due to the nature of the game, transitions between states are deterministic. With the large state space of our model and the exclusion of the explanation of the full rule set of the game for this paper, all transitions will not be described. A few examples of transitions that occur in the game are:

- after any roll of the dice, each player accrues the resource from the any hex tile with that rolled number on it once for any settlement that they have on a vertex the hex tile, and twice for any city that they have on a vertex of the the hex tile and the number of accrued resources is added to that players resources.
- after a player decides to build a road on a specific path and they have sufficient resources, that road is then added to the board and path list, and the cost of a road, one brick and one lumber is removed from the player's resources.
- after a player decides to trade four, three, or two resources in (given that the trade is valid given harbors they are on) the resources traded in are removed from their player resources, and one of any resource of their choosing is added to their player resources.

For a full list of all potential transitions, please see [2], ignoring those relating the development cards and the robber.

### D. Rewards

The scoring of the game is decided by Victory Points as discussed in previous sections. The rewards for reaching terminal states of the game differ for each algorithms implementation, and will be discussed in later sections.

## IV. Solution Approach

### A. Random Player

We first wanted to create an autonomous baseline player, one that would complete the game at a minimum. The easiest way to accomplish this was to create a random player who would choose every action at random. As such, this player chooses their initial settlements at random from all possible valid intersections, chooses their initial roads randomly, as well as game loop actions, building actions, and trades. This would give us a baseline to compare against for all other created players moving forward.

### B. Simple Heuristic Player

For this heuristic player, we wanted to create a player that would play, emphasizing gaining victory points quickly. One part of this came from the choice of initial settlements. We went back and forth on the optimal strategy, but eventually concluded that the best position for the initial settlements of our heuristic player were the locations which yielded the most expected resources (i.e., the intersections with the largest probability of dice rolls). With the ability to trade, this allowed for our player to receive any resources which they may need at any given time in the game. As for the main game loop, as stated before, the heuristic player wished to gain victory points as fast as possible. The main avenues for this are by building cities and settlements. Given the resources that the player had at any given turn, we would compute the available actions with those resources, then first try and see if one of those actions allowed for the building of a city, then a settlement, and if neither of those were possible, we would try and build a road. Finally, if those options were unavailable, we would try and trade in resources in order to enable a building action to be possible. Due to the required resources for building, we would only ever trade in resources if we had less than 3 grain, less than 4 ore, or if we had 0 of any of wool, brick, or lumber. This allowed us to guarantee that if we made it through all conditional statements in the trading action, we would be able to build something at the next action. This methodology worked quite well, with one drawback being non-optimal decisions for the placement of new buildings. The decisions for where to build anything were random, and as such, this player would not make the strongest moves at each decision.

### C. MCTS Player

When the game is started, the MCTS player chooses their initial settlements and roads the same as the heuristic player. This was done primarily due to the difference in strategy and state space before initial settlements are placed. Because the board has 52 places to place a settlement, using a tree search to iterate and search through all these options would be extremely intensive to gain any value out of. This is vastly different from the typical game function when the initial settlements are already placed, where there not as many options to place settlements as they must be expanded upon from the existing buildings the player owns. Additionally, from our own personal experience from playing Catan, it's often true that the first settlements are placed in intersections with the highest probability of dice rolls, as done with the heuristic player. With all this in mind, we decided it was best to keep our placement of initial settlements to be done heuristically.

Our Monte Carlo Tree Search implementation framework was taken from an existing python package built to more efficiently build a tree structure, search the tree, and perform rollouts [7].

This framework revolves around a "MyState" Class used to track functions to access actions, next states and rewards. When initialized, the class takes in a copy of the game state to allow making moves on the current board without affecting the actual instance. The following main functions were instituted to simulate MCTS with Settlers of Catan:

- 'get_possible_actions' would return all possible actions for the MCTS player to take based on it's current resources. Due to function built into the game to check if the player had valid resources and a valid building location, this made it easy to get a list of what was possible from this game state.
- 'take_action' would return a new state after taking a given action. This function was pivotal in transitioning from one state to another based on a player's action. For any action besides passing their turn, this function would take said action and return the game state while maintaining the same player's turn. This would also return the state having the same player's turn ongoing, as player's turn is not over until they pass to the next player. As such, taking the pass action would preform a rollout of the other three player's simulated turns, and return when it was the MCTS player's turn again. Thus, this function allowed MCTS to explore possible futures by simulating actions from the current state.
- 'is_terminal_' would return if a state was terminal, only in the case that one of the players in the game accumulated 10 or more victory points. 'get_reward' would return the reward for the current state. We returned a very large positive reward if the state was one where the MCTS player had 10 or more victory points, a small positive reward if the previous action was one that increased victory points (building a settlement, building a city, or getting longest road), and a very small negative reward if the state was neutral and no victory points were gained. These two functions were important as each node contains statistical information about the outcomes of simulations that pass through it. Each node provides a score reflecting the desirability of that state, essential for backpropagation in MCTS where the simulation results can refine the strategy.

This MCTS implementation allowed us to handle how state transitions were performed as well as the values of each state. Additionally, we were able to control the depth of the rollout in terms of number of iterations or cap the time of the tree search to prevent unnecessary searching.

### D. Deep-Q Learning Player

The first challenge in making the DQN agent was formulating the environment in such a way that constructing the neural network architecture would be easy. We decided to use the Gymnasium [5] library. Gymnasium is an open source Python library for developing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments. As we had a custom environment we used a Gymnasium wrapper in which we formulated the state space, the action space, a function to get state observations, a step function to take a step in the environment, a reset function, and then a reward function.

The state space was constructed as three one dimensional arrays and one two dimensional array. The 1-D arrays represent the "vertex state", "edge state", and "victory points" while the two dimensional array represent the board itself. The "vertex state" is of size (54,) which is all of the possible hex tile intersection points in the game. It contains a +1 or +2 for the agent's own settlement or city respectively as well as a -1 or -2 for adversarial settlements or cities respectively. The "edge state' is of size (72,) which is all hex tile edges in the game where one could possibly place a road, it contains a +1 or -1 for the agent's or an adversarial road respectively. The 2-D array representing the board itself is of size (19,6) there are 19 hex tiles in the game and 6 possible resources: "forest", "hills", "pasture", "fields", "mountains" and "desert". For each hex tile (row) the array contains which resource it is (column) and the roll number of that tile (entry) so this array has one non-zero value per row.

The action space is four 1-D binary arrays. The first is of size (72,) called "build roads" representing all hex tile edges, a 1 indicates build a road on the edge (index) where the 1 is located, a 0 indicates do nothing. The second is of size (54,) called "build settlement" and again a 1 indicates build a settlement on that intersection. The third is of size (54,) called "build city" with the same idea. The last is of size (75,) which represents the maximum possible number of trades that a player could possibly have options for in a game of Catan. Again a 1 indicates to take that trade, a 0 indicates do nothing. For each action there is also a validity check using a valid action mask to see if the build or trade is valid.

After defining the state and action space we next needed to formulate the get observation, step, and reset functions. The get observation was pretty simple it involved dissecting the game object to fill up the arrays defined above. The step function first loops through all the other players in the game allowing them to take their turns and then goes through the 1-D action arrays defined above building and trading where indicated. The reset function creates a new game object, and then goes through the building phase heuristically, in the same way described in the heuristic player section. Finally, the reward function was an outcome placed based reward. The agent got a +100 for winning a game, a +20 for getting second place, a 0 for third place and a -50 for last place.

Now, we were able to register our environment, make a Catan environment object and construct out DQN model. Gymnasium has built in flattening functions so that we could flatten both our state and action space to input into our neural network. We made the DQN network from scratch using tensorflow for the architecture. The parameters of which follow. After flattening the input later (state) was of size 244 the hidden layers were of size 512, 256, 128, 64 which then mapped to our actions which after flattening were of size 255. The activation functions were LeakyReLU except for the last layer which was a binary step function centered around 0.5 in order to make the output a binary array. The replay buffer size was 1000, we update the target q network every 25 games, and we use an epsilon greedy policy starting with epsilon = 0.25 but decaying each turn with a decay rate of 0.95.

## V. Results

### A. Heuristic Results

In order to validate that our heuristic strategy was stronger than random, we ran simulations of one heuristic player versus three random players for 1,000 trials. We found that over these trials, the heuristic player won over 80% of games, as can be seen in Fig. 2.

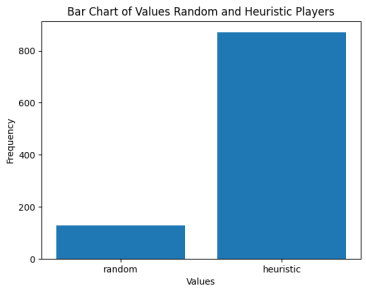


Fig. 2. One Heuristic vs. Three Random Players Win Distribution.

When further analyzing these games looking at the average number of turns which occurred in a game with one heuristic player versus three random players, we found that the average number of turns was approximately 125 as can be seen in Fig. 3.

These results were satisfactory for us proving that our heuristic player performed at an elevated level to that of a random player. Additionally, a traditional game of Settlers of Catan takes 71 turns, and with our reduced game which had limited capabilities, rolling a seven would not do anything, we found our average number of turns of approximately 125 to be reasonable [3].

### B. MCTS Results

In Fig. 4 we can see a simulation of 25 games versus one Heuristic player and 2 Random players. The MCTS player

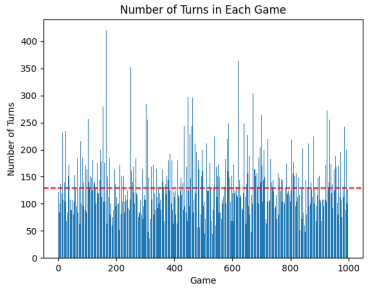


Fig. 3. One Heuristic vs. Three Random Players Number of Turns.

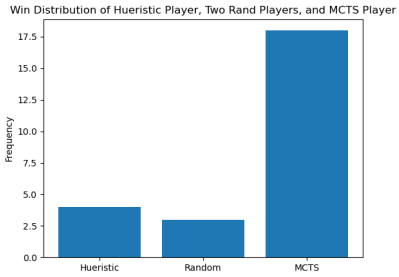


Fig. 4. MCTS Player Wins vs. 2 Random Players and 1 Heuristic

wins outright in 18/25 or 72% of games. We can see that the MCTS player performs consistently well against a singular heuristic player.

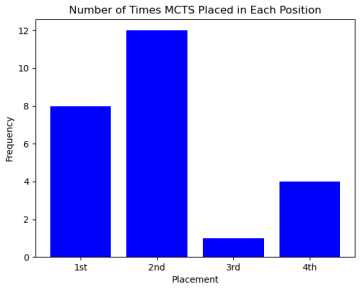


Fig. 5. MCTS Player Placement vs. Three Heuristic Players

In a simulation of 25 games versus 3 heuristic players we can see from Fig. 5, we can see that the MCTS player placed in 1st or 2nd place in 20/25 games, or 80% of games. Although is its not consistently beating all three players, is is performing well in terms of placement versus the other three players.

In Fig. 6 we can see the victory points of the MCTS player versus the average victory points of the other three heuristic players in each game. We can see that the MCTS player has equal or more victory points than the average of the heuristic players in 14/25 games.

### C. DQN Results

First we will take a look at how the DQN agent stacks up against 3 random players.

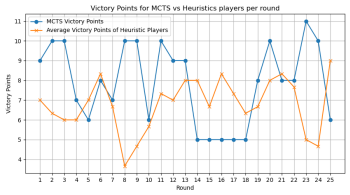


Fig. 6. MCTS Victory Points Per Rounds vs. Three Heuristic Players Avg VPs

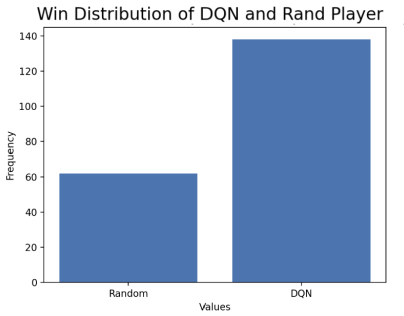


Fig. 7. Win Distribution of DQN and Three Rand Players

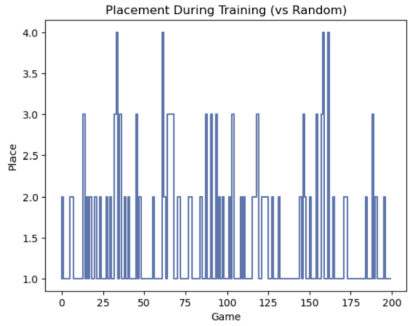


Fig. 8. Placement of DQN Agent vs. Three Rand Players

From Fig. 7 we can see that the DQN Agent wins around 2/3 of the time against three random players during training. Fig. 8 shows us that even when the DQN agent doesn't win against random players it consistently will get second place. One peculiar note with 8 is that this figure was generated during 200 training episodes. If the agent was learning over time we would expect its placement to significantly be higher over time, instead it appears to be somewhat evenly distributed throughout training. Next we will add one heuristic player to see how the agent performs.

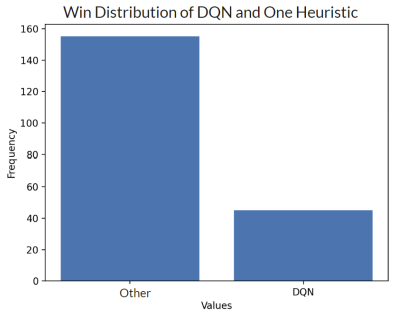


Fig. 9. Win Distribution of DQN Agent vs. Two Rand Players and One Heuristic Player

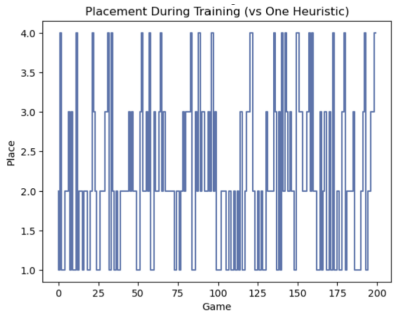


Fig. 10. Placement of DQN Agent vs. Two Rand Players and One Heuristic Player

From Fig. 9 we can see that the DQN Agent does significantly worse against one heuristic player than it does against three random players. It only wins around 1/3 of the time. Fig. 10 shows that it will rarely get last place, but once again we see very little evidence of this agent learning over these 200 episodes. More on this later. Finally, the following figures show the DQN Agent's performance against three heuristic players.

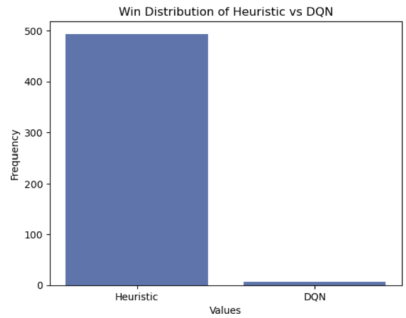


Fig. 11. Win Distribution of DQN Agent vs. Three Heuristic Players

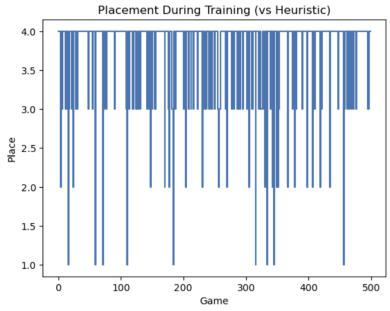


Fig. 12. Placement of DQN Agent vs. Three Heuristic Players

From Fig. 11 we can see that the DQN Agent does much worse against three heuristic players. It only won 9 games out of 500 during training. Fig. 15 shows that it will rarely get first or even second place, but once again we see very little evidence of this agent learning over these 500 episodes. This was the case every time we trained the DQN agent. A couple thoughts as to why the DQN Agent might not be learning significantly over time. First, games take around 1-2 seconds per game so it is difficult to train many games. Secondly, the action space is really big it is a 255 sized binary array, without many episodes to train on it is really hard for the DQN agent to learn within this architecture. If we were to continue this work we would want to find a way to reformulate the DQN architecture to lower the dimensionality of the action space significantly.

### *D. Results Conclusion*

From our trials with both the MCTS player and DQN player, we were able to determine that the heuristic player is really good in this reduced action space. Without development cards or the robber, the game is much simpler and can be expressed as an optimal policy much easier. As such, the heuristic player is very close to an optimal building and trading policy agent so it is difficult to beat.

With the added use of development cards, agents could find new ways to get resources and victory points, expanding their options for a path to victory. A robber would allow an agent to methodically place it in a location that is hurting a strong foe. With both of these changes, we could likely see a biggest difference between our created agents and the heuristic player.

## VI. Conclusion

From our testing we found that our MCTS player was able to win 32% of games versus three heuristic players, while our DQN player was able to 1.8% of games. While the DQN results were disappointing when it played against all heuristic players we have some ideas on why it was under-performing, the most significant being the huge dimension of the action space These finding lead us to conclude that MCTS was significantly our best AI agent with a performance on par with if not slightly better than a heuristic player.

If given more time we would like to conduct further research into reward shaping, investigating whether starting position

matters, lowering the DQN action space dimension, as well as expanding our algorithms to be able to play given all elements and phases of a full game of Settlers of Catan. Finally we'd like to explore how to combine our methods of DQN and MCTS as is done in AlphaGo and other successful RL game agents [1].

## VII. Contributions and Release

- Max Gerber - heuristic player tuning and testing
- Mitch LaRocque - MCTS integration and testing
- Maxwell Van Sickle - PyCatan/DQN/Gymnasium integration and testing

The authors grant permission for this report to be posted publicly.

## Acknowledgment

M. M. M. thanks Professor Zachary Sunberg for his teachings and guidance throughout the semester and as well as specifically this project.

## References

[1] J. Hu, F. Zhao, J. Meng and S. Wu, "Application of Deep Reinforcement Learning in the Board Game," 2020 IEEE International Conference on Information Technology,Big Data and Artificial Intelligence (ICIBA), Chongqing, China, 2020.

[2] Catan Game, "Game Rules," Catan, 2024. [Online]. Available: https://www.catan.com/sites/default/files/2021-06/catan_base_rules_2020_200707.pdf.

[3] Cates, A. (2022) "Board game breakdown: Settlers of Catan, the Basics," Alex Cates. Available: https://www.alexcates.com/post/board-game-breakdown-settlers-of-catan-the-basics.

[4] D. Shah, Q. Xie, Z. Xu "Non-Asymptotic Analysis of Monte Carlo Tree Search," [Online]. Available: https://arxiv.org/pdf/1902.05213.

[5] Farama Foundation, "Gymnasium", GitHub, v1.0.0 2024. [Online]. Available: https://github.com/Farama-Foundation/Gymnasium

[6] J. Waller, "PyCatan2," GitHub, v1.0.1, 2021. [Online]. Available: https://github.com/josefwaller/PyCatan2.

[7] K. Strümpf, "MCTS," GitHub, v2.0.5, 2024. [Online]. Available: https://github.com/kstruempf/MCTS.

[8] K. Vombatkere, "CatanAI," GitHub, 2020. [Online]. Available: https://github.com/karanvombatkere/CatanAI.

[9] V. Mnih et al. "Playing Atari with Deep Reinforcement Learning," [Online]. Available: https://arxiv.org/pdf/1312.5602.

## VIII. Appendix

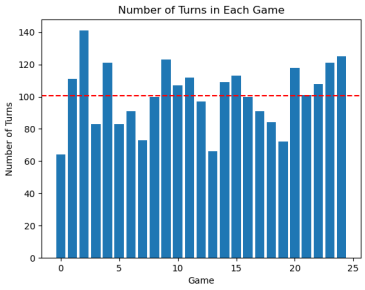


Fig. 13. Number of Turns per Game of MCTS Agent vs. Three Heuristic Agents

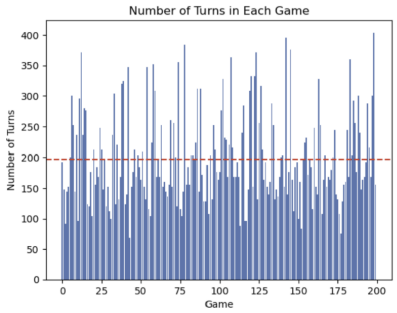


Fig. 14. Number of Turns per Game of DQN Agent vs. One Heuristic Player

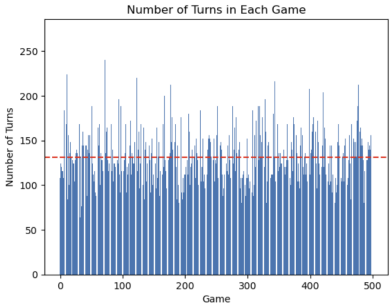


Fig. 15. Number of Turns per Game of DQN Agent vs.Three Heuristic Players