# AA228 Final Project: Settlers of Catan Simulator

John Spencer, Hillary Umphrey, Nikita Lilichenko

November 9, 2020

## 1 Abstract

In our project, we created a simplified version of the board game Settlers of Catan that involves only one player and then trained a AI agent to win the game as quickly as possible. We trained this AI agent using Q-Learning with a epsilon-greedy exploration strategy. With a suitable number of iterations during simulation and training, the AI agent began to learn high level strategies in our complex game space and outperform a random agent.

## 2 Introduction

Catan, previously known as The Settlers of Catan or simply Settlers, is a multiplayer board game designed by Klaus Teuber, and was first published in 1995 in Germany. Players take on the roles of settlers, each attempting to build and develop holdings while trading and acquiring resources. Players gain points as their settlements grow; the first to reach a set number of “victory” points, typically 10, wins. It is popular in the United States where it has been called ”the board game of our time” by The Washington Post. Richard Dansky, a famous game designer, comments that ”for all of its elemental simplicity, The Settlers of Catan has breathtaking depth and breadth of experience. It’s a resource-management game, defined by position and strategizing. It’s a social game, defined by [trading] of resource cards ... It’s a game of chance, ruled by dice rolls and card draws. It’s a hardcore game and a light social pastime and everything in between, a laboratory where I can test a hundred different play styles and a genuine reason to invite friends over” [1]. As illustrated in the quote, the world of Catan provides a perfect opportunity to explore the uncertainty involved in decision making. In our project we will use reinforcement learning, a subset of the machine learning techniques, to train a smart AI Catan player through trial and error.

## 3 Related Work

In the report ‘Q-Learning for a Simple Board Game’, Arvidsson and Wallgren explored applying Q learning to the board game dots and boxes and their conclusion was that a higher discount factor always gives better results. Furthermore when training a new agent, a low learning rate is more accurate, while training a new agent against an experienced agent allows for a higher learning rate [2]. In a domain specific application of RL, ’Reinforcement Learning of Strategies for Settlers of Catan’, Michael Pfieffer tried four different types of hierarchical RL strategies: feudal (high level policy selecting a new behavior at every time step), module-based (high level policy selecting new behavior every time a sub-goal is reached), heuristic-based (hand coded high level policy with small decision behavior learned by self play), and lastly a guided approach (where heuristics are introduced early on during training, but then the new policy relies on its own experience at the end). In his results, the feudal approach resulted in over fitting because the agent changed high level strategies every turn, while the module-based agent fared better and was more competitive against a human player by making more human-like small decisions, but still had an issue with switching high level strategies too often. The heuristic approach was the strongest as it learned similar human-like behaviors for small decisions, but did not suffer from missing sight of a consistent high level strategy, whereas the guided strategy could not gain the same effective small decision making. In our project, we focused on self-play as our main action of learning in order to emulate the high performing heuristic based approach. [3]

# 4 Approach

We constructed the code for this Settlers of Catan Simulator in several major steps:

## 4.1 Constructing Game Board/Initializing Values

The first step was to establish the play board and initialize the values and rewards of the associated tiles. Understanding the computational limits of what we can achieve for the scope of this project, we signficantly reduced the complexity of Catan. The board itself included a 3 x 3 tile space arranged in a square pattern; there were a total of 16 vertices of the squares, which served as the positions to build settlements, with edges connecting the vertices. The space within a particular square (enclosed by vertices and edges) was considered the tile, and had a resource of either “wood” or “brick”, each with its own reward yield. Notice there are only two resources: this is because after testing our program we realized that a 9 tile board with 5 resources was too imbalanced for the agent to learn a consistent strategy (there were too few opportunities to gain enough of each resource), reducing the resources to only 2 and eliminating the need to buy settlements with resources created a better environment for learning. The relationship between all tiles and their related vertices was constructed with a dictionary data structure, while the tile resources and action space for building a settlement were constructed as arrays, where the index corresponded to the associated tile. In the context of our game, an action is defined as building a settlement on a vertex, and is represented as an integer between 0 and 15 inclusive, which corresponds to the index of the vertex on which to build a settlement.

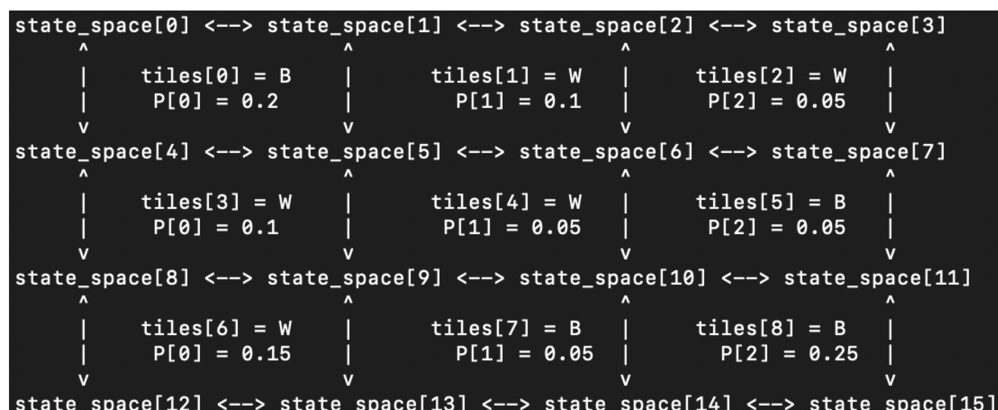



Figure 1: Game Board Representation

Once the board was constructed and populated with values, the next sub-step was to implement a game that made decisions to build settlements randomly. A state space, which describes the state of the game board at any time, is represented as an array of length 18. The first 16 elements correspond to the amount of settlements the player has on the 16 settlement positions. The 17th and 18th elements correspond to the total number of “wood” and “brick” resources the player has acquired, respectively. At the start of each new game, the state space is initialized to [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], as the player does not currently have any settlements on any positions nor any resources. For each turn of a certain amount of game turns, an action was randomly selected using the function $random.randint(0, 15)$, then the state space was updated with the associated action and resources based on the neighboring tile to deliver a reward for building a settlement there. The total reward was calculated as a cumulative of all of the rewards received from all the turns. Implementing this simple random policy allowed us to simulate a game with any amount of turns and receive rewards completely independent of tiles and their potential yields.

## 4.2 Constructing the Q-Learning Algorithm

The second major step in creating the Catan simulator was split up in to two key parts: constructing and implementing a working reinforcement learning algorithm that, on each turn, could pick an action based on

the maximum reward that the player could achieve given the current state space of the board, rather than picking a random action to take. For this particular project, we decided that the Q-Learning algorithm would be the best algorithm to use, given its simplicity and relevance in regards to the data that we generate about states and actions every turn. With the Q-Learning algorithm, we would be able to generate a state action matrix, which would house information about taking a particular action while the game board was at a particular state, and this crucial point of reference would be able to guide the player's decisions to maximize reward at every turn. In order to get substantial game data to populate the state action matrix with maximized rewards, we would run a set amount of game simulations before actually playing the game, then run the final game with that matrix at the player's disposal.

### 4.2.1 Constructing the Q-Learning State Action Matrix using Simulations

Constructing the Q-Learning algorithm was centered around the creation of the state action matrix. The highest level function responsible for building this matrix was called $buildQMatrix(action_space, initial_iterations, turns)$, which passed in the action space, total amount of simulated games to run, and the amount of turns to play per game. The first step of this function was to initialize the matrix with the first state space of all zeros (the size post initialization was 1 row (state space) by 16 columns (actions). Furthermore, a dictionary was initialized, where the key corresponded to the array representation of the state space, and the value corresponded to the row number of the matrix that represented that state space. This crucial data structure was created to ensure that there was always a correct reference to any row of the matrix, as the dictionary would always be updated along with the matrix. The discount factor and learning rates were also set in this function as well. This function then included a nested for loop; the outer represented running a number of games, while the inner represented running a number of turns per game. During each turn, the best action was determined with the function *selectBestAction*(...), a simulated game turn was played after which the reference dictionary was updated, and the state action matrix was updated at the appropriate state space row and action column with the reward generated from the simulated game turn. The entry in the matrix was updated with the following Q-Learning formula: $Q(s,a) \leftarrow Q(s,a) + \alpha * (r + \gamma * max_a(Q(s', a')) - Q(s,a))$, where s is the old state space, a is the action, s' is the current state space, a' is the new action selected, $\alpha$ is the learning rate, and $\gamma$ is the discount factor. This process encapsulated a bulk of the Q-Learning algorithm.

A significant aspect of the Q-Learning algorithm was the selectBestAction(Q, state_dict, state_space), which passes in the state action matrix, the reference dictionary, and the current state space. This function used epsilon-greedy exploration to make initial decisions without previous data given a certain epsilon value and its associated decay value. If a random value came back less than the epsilon value, the e value would decrease at the rate of the decay value, and a random action would be selected. If the random value was greater than e, the action would be determined by the highest reward within the state action matrix at that state. If the state space was not already present as a row in the matrix, the ideal action corresponding to the nearest neighbor to the current state space would be selected.

### 4.2.2 Nearest Neighbor Approximation

As our problem includes a massive state space we anticipated that during training there would be many states we would encounter that were not already present in the matrix. It is then crucial that we have a sound method for finding the already existing state space most similar to this missing state space. In initial versions of our training we compared each matrix by simply looked at the sum of differences between each element, or in our case settlement location. While this method does a decent job at finding matrices with elements at the same locations, beyond that there is no benefit offered for "similar" locations. Looking at the layout of our board a settlement on location 11 has much more in common with a settlement on location 15 than location 12 as 11 15 both reap a reward when tile 8 is chosen. Using our original vector subtraction method there would not be a distinction between these two. We pivoted to include a location based system that factors in proximity on the board. In this location based method we place the board on a Cartesian plane with location 12 (the lower left vertex) as (0,0). We then find the weighted average of all settlements on the board in the given state space. Once this weighted average is calculated for all state spaces we determine the nearest neighbor by calculating which neighbor's weighted average point is closest to that of the missing state.

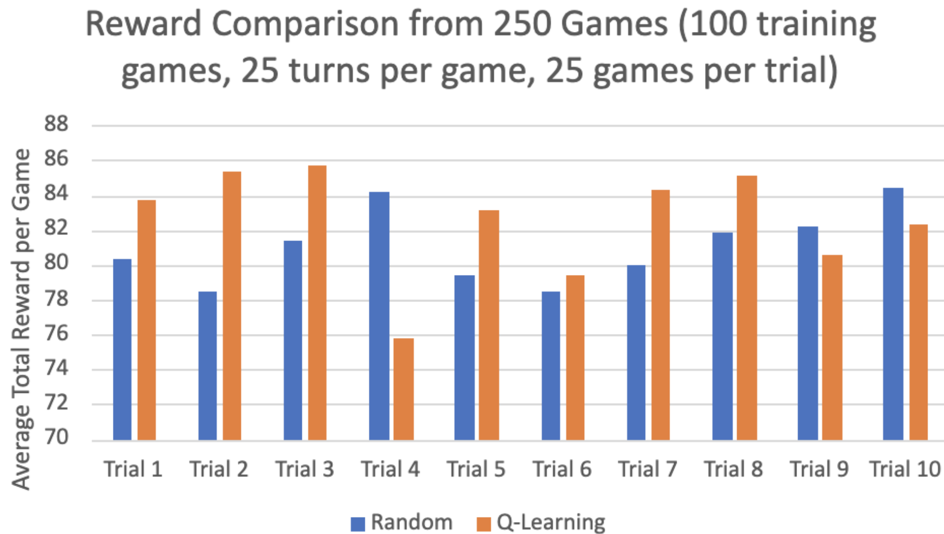
### 4.2.3 Implementing Q-Learning into the Final Game

After the state action matrix was completed and previous data was collected, it was time to actually play the Catan game. Similar to the original random policy game that we had implemented before, our new final game used the same system for running a set of turns, actually building the settlements (updating the state space array) and finding the cumulative reward for the game. The biggest difference between the original random policy simulator and the new Q-Learning simulator was the way the action is selected. During each turn, the function $decideOnSettlement(Q, state_dict, state_space)$ is called, and passes in the state action matrix, reference dictionary, and current state space. If a current state space already exists in the matrix and dictionary, its associated row is searched using the dictionary and the action with the highest reward at that state is returned with the $argmax()$ function. This action is returned and used by the remainder of turn. If a current state space does not exist in the matrix and dictionary, then the state's nearest neighbor in the matrix/dictionary is used instead. This action is then returned.

## 4.3 Random Policy vs. Q-Learning

After fully constructing the Q-Learning algorithm, the last step was to test whether or not the Q-Learning algorithm truly yielded a higher reward than a random policy. In order to get an idea of this hypothesis, the random policy and Q-Learning simulators were each run a set amount of times with a set amount of turns, while keeping all hyperparameters constant. The average reward was calculated for each of these algorithms, and the average was compared to determine the true victor.

# 5 Results




When testing our policy there are three mutable variables for each trial. Firstly, we can alter the number of training games the program completes while building the policy using Q-Learning. This is represented in our program as number of 'Simulations.' Secondly, we can alter the number of turns played in the training and testing period. Lastly, to speed up the testing phase the program runs test games in blocks, averaging the results of the block of games using a single optimized policy. The first round of testing was performed with a state action matrix created using 50 simulations. In these 250 trial games the random policy out performed the Q-learning policy. We predicted that this was due to the sheer size of the state space, causing only 50 simulations to not properly fill the state action matrix. The second round of testing was performed with a state action matrix created using 100 simulations. In this round there appeared to be a slightly better

| | Random | Q-Learning | Difference | % Improvement |
|---|---|---|---|---|
| Trial 1 | 80.35 | 83.75 | 3.4 | 4% |
| Trial 2 | 78.44 | 85.4 | 6.96 | 9% |
| Trial 3 | 81.4 | 85.64 | 4.24 | 5% |
| Trial 4 | 84.2 | 75.76 | -8.44 | -10% |
| Trial 5 | 79.44 | 83.08 | 3.64 | 5% |
| Trial 6 | 78.52 | 79.44 | 0.92 | 1% |
| Trial 7 | 79.96 | 84.28 | 4.32 | 5% |
| Trial 8 | 81.9 | 85.08 | 3.18 | 4% |
| Trial 9 | 82.24 | 80.52 | -1.72 | -2% |
| Trial 10 | 84.4 | 82.28 | -2.12 | -3% |

Figure 2: Trial Results comparing Q-Learning policy and a Random Policy

outcome using the Q-learning policy. For the 250 games the Q-learning policy outperformed the random policy by, on average, 2%.

## 6 Future Work

Looking forward there is much more exploration that can be performed using this simulation as well as a more complex version that more closely resembles the original game of Catan. Further investigation could examine the relationship between Q-learning policy efficacy and number of turns played. Similarly, more trials could be run with different combinations of the four hyper-parameters. Beyond these simple experiments, performance could have been improved with added hand written heuristics.

## 7 Work Distribution

John implemented the game simulation and made decisions regarding the simplification of the game. Nikita and Hillary decided on the learning strategy and wrote the code for the q-learning section. After the initial run through Nikita focused on debugging and Hillary focused on improvement of the approximation section. All worked on write-up.

## 8 Works Cited


1. Wikimedia Foundation. ‘Catan. https://en.wikipedia.org/wiki/Catan
2. Oksar Arvidsson and Linus Wallgren. ”Q-Learning for a Simple Board Game” Bachelor of Science Thesis. Stockholm, Sweden 2010.
3. Pfeiffer, Michael. ”Reinforcement learning of strategies for Settlers of Catan.” Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education. 2004.