# CS 221: Final Project Report

Authors: Sebastien Goddijn, Will Lauer, Noah Jackson

November 16, 2018

# Task Definition

The task that we decided to do for our 221 project is something which is close to our hearts, as we have all played Catan many times in the past, although none of us are particularly good at it. We set out with a very specific goal in mind: to create the best artificially intelligent Catan player that we could. We wanted to make an AI that could play Catan as well as we could, and potentially even beat some of the better players in our residences, so not a system that could beat the world champion of Catan, but one that could effectively beat an average player who knows what they're doing.

# Infrastructure

In order to train our player we needed to create a fully functional implementation of the Catan board game, that allowed both artificial and human players to play against one another. In our implementation we divided the game into several modules, the most important of which we will outline below.

**play.py**:
This is the primary point of entry for running a game. This module initializes the board, the players, and the display, then enters our main game loop. After running the first two turns, in which each player has the option to place two settlements and two roads, it loops until a victory condition is met for some player. At each turn, resources are distributed amongst the players according to their settlement and city locations. After that, the logic branches, depending on whether the current player is a human or an AI. If the player is human, the GUI interface is used to receive a player's action: a click on the board prompted by text in the terminal. If the player is an AI, all possible moves from this state are collected, and the AI picks the most valuable action evaluated using whatever weighted feature extractor it is running on. After this point, the game state, board state, and display are all updated, and the game to proceeds to the next turn.

**game.py**:
This class handles the state of the game. We defined this module to track updates to the game that all of the players need to access. When an AI needs to choose a move, game.py is what it queries to receive possible actions, resource values, and all such necessary values. All game logic for valid settlement locations, road locations, and purchasable items is handled here. In addition, it serves as an intermediary between the players and the display. For example, if a player moves the robber, they will call setRobberLocation in game.py, which then takes care of updating the game state and the display. By abstracting away this functionality, we have managed to avoid most discrepancies between display and actual game state, making for far easier debugging and testing.

**player.py**:
This is divided into two classes which share the super-class of Player. The superclass defines the methods that both human players and AI players have in common, like placeRoad, placeSettlement, and discardResource, whilst the subclasses are more specific to the type of player.

The first sub-class is the HumanPlayer, which defines all of the logic that is specific to human players. We use text-based inputs to allow users to pick the move they want to make, and then make them click on the display to place items on the board.

The second sub-class is the AI Player, which is the superclass for our two AI implementations. The first is our weighted AI, which serves as both our initial training method and also the baseline on which we can test the performance of our other AIs. It gets assigned random weights for each feature in our feature extractor, and chooses its moves based on these weights. It then updates its weights towards the weight vector of the winner at the end of every game, which allows it to improve its performance. We use an untrained weighted AI (with random weights) to provide a baseline on which to test our other AIs, and we use a trained weighted AI as an improvement on this naive implementation.

The second AI implements a Q-learning algorithm which updates its weights at the end of every turn, as opposed to the end of every game, and also includes adversarial features in its feature extractor, which our basic weighted AI doesn't.

**display.py**:
Handles the GUI used by human players. After any player, human or AI, makes a move, a method in this class corresponding to their action is called in game.py. The board display is updated accordingly. Running the display separately from the primary game class abstracted away a lot of the gritty image-manipulation details, and allows us to save processing time by updating the display exactly when an action takes place, rather than doing a lot of busy-waiting for something to happen.

**devcards.py, node.py, board.py**:
We maintained an object-oriented approach to the design of the whole system such that we could pass instances of objects which stored all of their current information, and their functionality, and thus allowed good decomposition of our code.

The devcards.py file deals with the creation and playing of development cards in game. Each development card keeps track of its type, the player that owns it, and also its .play() function, which handles the logic involved with playing that card. As such we did not need to handle much logic in play.py but simply got the instance of a devcard and told it to play itself.

The node.py file was one of the most important structures in our implementation, and we

centered most of our game-playing logic around it. It defined the board set-up (which ended up being just a matrix of Node instances), and had access to all of its neighboring tiles, which player owned it, what piece it was occupied with, and its position on the board. This meant that whenever a player needed to build a road, or collect resources after a roll, we could simply query all of the Node instances that they had occupied and determine the right course of action from the information that provided.

In board.py we kept track of the current game state, as it kept track of every defined instance of nodes, which we would then update in game as their states changed. This meant that rather than having a large number of getters and setters, we used a standardized coordinate system based on nodes to access the node object itself, which we then modified. This allowed us to minimize our front-end code, while maintaining flexibility. All of the conversion from node coordinates to tile coordinates, etc. is handled in our back-end code, allowing us to use a simplified object-oriented interface when working with the board and game state.

**test.py**
We have a number of different testing files which we used to train our AIs. To do this we initialized weights files at random on the first training iteration, and for every iteration after that, at the end of the game, we would write our updated weights vector the weights file, and read it in from the same place at the start of the next game. This allowed us to automate a huge number of trials, and also allowed us to determine win percentages between different players over a large number of trials, which was useful for data generation and error analysis.

# Approach

**Intrinsic Challenges**
The challenges involved in building this system had to do with the number of choices that were available at each turn, and attempting to develop an optimal strategy to pick moves given the current game state. Although the primary target of the game is to collect as many victory points as possible, there are many factors for a player to consider. They need to ensure that they have appropriate access to resources in order to build structures that give them victory points, they needed to understand when to purchase development cards, when to exchange a resource that they had a large quantity of for a different resource that would allow them to advance their position in the game, and even making sure that they kept the number of resources in hand under a certain threshold in case a 7 was rolled. All of this in tandem with the fact that there are adversarial moves that can be made to jeopardize a players position, for example moving the robber to prevent their resources from being distributed, or blocking off one of their roads. As such we needed to build a system which allowed us to determine the state of all of these features, which is exactly what we did, as outlined in our infrastructure.

**Baseline**
Given the large number of possible moves and different strategies to consider, we felt that a purely random AI, that picks moves at random based off of its possibilities, was a little bit too basic to test the effectiveness of our implementation. This led us to a decision point, where we chose between either creating a weighted AI with randomized weights, that would pick moves based off of an evaluation function, or using a weighted AI with weights we initialized ourselves. We once again felt that randomizing the weights could lead to an overly naive baseline, so we eventually decided to create the weights ourselves. For example we know that increasing your score is important, so we put a relatively high weight on the score feature. Additionally having the longest road or the largest army is usually very valuable in Catan so we gave these features relatively high weights as well. In this way we generated an AI that approximately represented a naive human player, which would serve as a baseline.

**Oracle**
We wanted the Oracle to be reasonable in the scope of our project and as such our Oracle is simply a measure of whether our AI can beat a good human player. This is someone who's played extensively and as such has built an intuition and a strategy for the game of Catan, but isn't someone who competes at a higher level or anything like that. This would mean that our system has become approximately as good or potentially even better than a human which we believe is a large enough test of the AI that we manage to train.

**Methods**
The first method we used was Q-learning, however we had two different iterations of this method in our two different AI's. In our first AI implementation, which we call the weighted AI, we used the Q-learning algorithm only once, at the end of every game. We wanted to generate a number of AI's with randomized weight vectors for each feature, and then we would evaluate potential moves by multiplying this weight vector with the features associated with doing a possible move, and chose the move with the highest score at the end of this. Then, at the end of the game, we would use the Q-learning algorithm to update our weight vectors, and used the winning players weight vector as a target. This has some benefits, as after enough iterations the weight vectors converge to seemingly optimal values, however these optimal values are completely dependent on the initial randomized weights. One way we attempted to reduce the importance of this initial randomization was to drop a player every 10 games and reset them to a random weight vector, and then re-introduce them to the game, which gave us slightly improved results.

The second Q-learning method we implemented updates weights at the end of every turn and the end of every game. This is done by defining prediction to be the eval(previous state), then defining target to be eval(current state). Updates at the end of the turn use very small $\eta$, as there can be many turns in a game.

To supplement the Q-learning methods applied above, we also built a more complicated evaluation function that uses a heuristic to estimate opposition moves given our players choice of move. This improved evaluation function is discussed in more detail later in our error analysis.

# Literature Review

We found a limited body of work that dealt specifically with the game of Catan, however the few that we did discover explored orthogonal techniques to ours with a relatively impressive level of success. The first technique we discovered was the use of monte-carlo tree search, which developed an AI that was able to perform quite successfully with a minimal amount of domain knowledge [1]. The second implementation built directly off the first, but incorporated the possibility for AIs to trade with one another, and focused specifically on developing a move pruning heuristic for the game [2]. We wanted to explore a contradictory pruning heuristic which instead of pruning specific moves avoided branching altogether by only evaluating optimal moves for each player, to determine whether this would be more or less successful than their implementation. Since every player had a different heuristic function, their perception of the other players behavior was not always consistent with the behavior itself, which we hoped would lead to interesting results. Additionally, we found no working open-source implementation for the game of Catan, and as such wanted to build our own infrastructure to put online in the hopes that future teams will be able to train on it and improve on the players that we have created.

# Error Analysis

To get the results for this analysis we firstly trained each individual AI by playing a total of 1000 games with four of them playing against each other, and learning from each others successes and mistakes. Then, we took the weight vector of the most successful AI (as per the number of wins it had in training), and simulated 100 games against our baseline, with our AI starting at a new position every 25 turns. This worked by creating three Baseline players with the weight vector that we previously mentioned, and asking them to play against the AI we wanted to evaluate, and recording the winner in each iteration of the game. To ensure that games ended, even if the board got into a deadlock, we capped the number of turns each player could have to about 200.

**Table showing results of weighted AI against Baseline**

| Turn number | Num Trials | Win Percentage |
|---|---|---|
| 0 | 11 | 100 |
| 1 | 8 | 100 |
| 2 | 7 | 100 |
| 3 | 6 | 100 |

The first trial we ran involved comparing our trained, weighted AI against the baseline AI which we decided the weights for. The weighted AI exited with a total win percentage of 100%. The issue in this case was that our trained AI had developed a strategy which mainly consisted of building a load of roads, which limited the possibilities of the other players in the game. As such, although the AI got a win percentage of 100% it was only able to finish

around 32 of the total 100 games that we wanted it to play. The issue with the weighted AI was that it only learned at the end of the game by attempting to alter its weights towards the winner. However, given that all the initial weight settings were randomized, it was highly likely that this convergence represented a local optimum and not a global optimum, which explains the unique results described above.

**Table showing results of our basic Q-Learning AI against Baseline**

| Turn number | Num Trials | Win Percentage |
|---|---|---|
| 0 | 25 | 60 |
| 1 | 22 | 77.2 |
| 2 | 25 | 80 |
| 3 | 24 | 91.7 |

The second set of trials we ran involved our basic Q-learning AI against our Baseline. This Q-learning algorithm was able to update its score at every turn by predicting the score it expected to have at its next turn, and then comparing this value to the actual score it obtained on its next turn. Then, once it was good at predicting what its score would be at the next iteration, it would select its weights to attempt to maximize this prediction, which we believed would grant far better results in terms of move decision making. Our results indicated to us that this AI was a much more legitimate type of player, which didn't clog the board-state or do anything otherwise adverse. Because of this it was able to complete a total of 96 out of the 100 games it started, with a total win percentage of around 77%. Although this seems less than the 100% that we got using our weighted AI we were far more confident in this AIs level of play, and this reflected a positive forward step in our training.

**Table showing results of Q-Learning AI with Win Percentage Target against Baseline**

| Turn number | Num Trials | Win Percentage |
|---|---|---|
| 0 | 25 | 84 |
| 1 | 24 | 87.5 |
| 2 | 25 | 60 |
| 3 | 25 | 88 |

Our third AI followed a method similar to the second, where used Q-Learning updates at the end of each turn, followed by a larger update at the end of the game. With this trial, rather than aiming to maximize our score, we aimed to maximize the probability given a particular score. This probability would be equal to 1 at the end of the game if a player wins, and 0 if they lose. At each other turn, this probability would be equal to the evaluation function of that given game state, which is a number between 0 and 1, unless the weights proved to be wildly inaccurate. Maximizing the probability of winning at a given game state performed slightly better than our basic Q-Learning across all starting turn positions; our theory is that maximizing the probability of winning rather than the expected number of points better represents Catan as a zero-sum game, and rewards reaching 10 points more

than our other methods. All but one of our trials finished without timing out, reflecting the more aggressive style of play expected of this method, and demonstrating further gains in our training method.

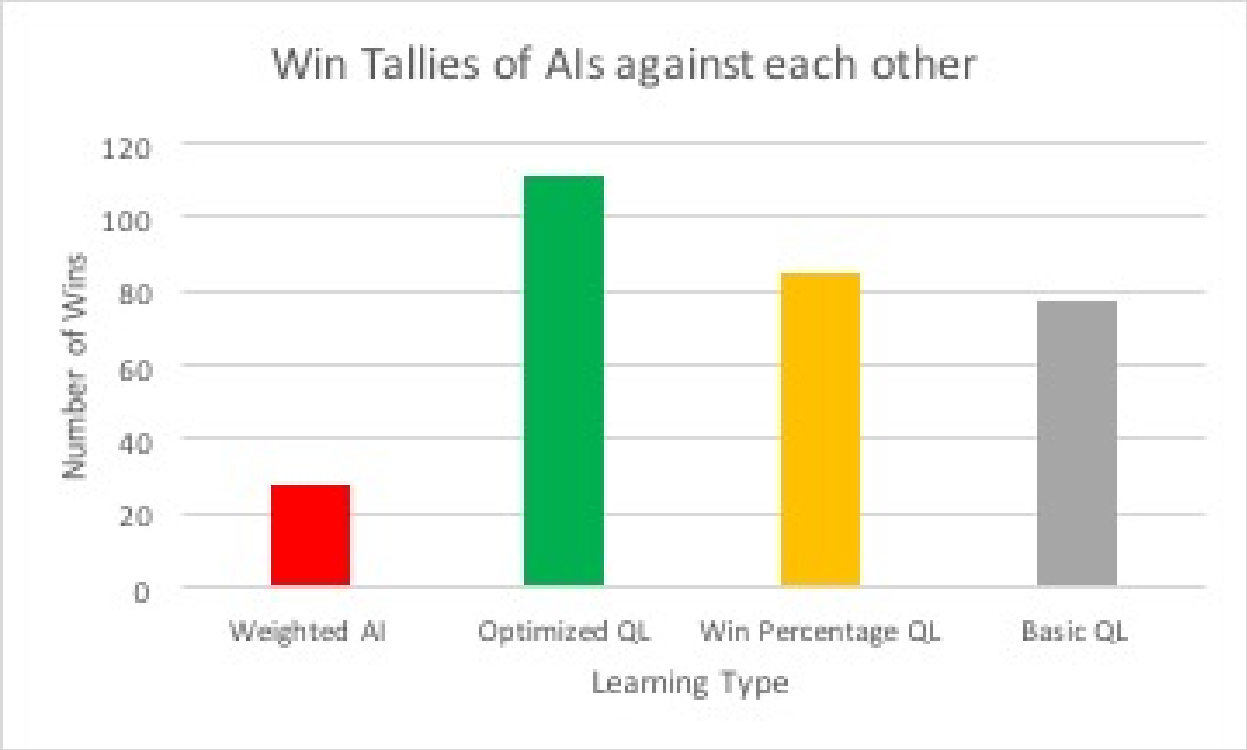
**Table showing results of basic Q-learning with Optimization against Baseline**

| Turn number | Num Trials | Win Percentage |
|---|---|---|
| 0 | 23 | 100 |
| 1 | 22 | 95.5 |
| 2 | 23 | 84 |
| 3 | 25 | 91.4 |

The last AI technique we implemented was our basic Q-Learning AI coupled with a more advanced evaluation function that considered future moves. We initially wanted to implement a minimax algorithm, but we realized that players in Catan typically focus on maximizing their own score, so minimax trees were a poor way to estimate future states. We then considered expectimax, but again we found that it wasn't necessarily a good way to model opponent behavior that focuses primarily on maximizing ones own score.

Each turn of Catan has many possible moves, as you can choose to buy variety of different pieces, then for each piece you buy, there are many locations where the piece can be placed. Evaluating the best position using only our weights and features is already relatively expensive, and this cost balloons quickly when considering the possible move combinations for 4 players. With these limitations in mind we instead decided to model opponent moves by estimating the move they would make using our players weight vector and feature extractor on the opposing players game state. This assumption is not necessarily correct, but it is very valuable as a heuristic because it allows us to avoid expensive branching factors that are inevitable when considering the possible moves of three opposing players. The reduced expense of using this heuristic allowed us to train on far more games, allowing us to get finely tuned weights while still considering the behavior of opponents.

**Graph showing results of AIs playing each other**




To see which AI was the most successful we create one of each type of player that had, and pitted them against each other for 300 games. As you can see from the above graphic, the training method which was the most successful was that of the Optimized Q-Learning, as it was able to win around 111 games, which gave it over a third of all the wins. In second and third, and relatively close to one another were the two other Q-learning methods, which performed almost equivalently to one-another, as we saw when they were pitted against the baseline test, and finally, in dead last, we had our weighted AI with only around 28 wins. This is what we expected when we considered how they performed against our baseline, and also shows a steady improvement of all the techniques that we ended up implementing.

# Appendix

Best Weight Vector for Weighted AI:
{"Year of Plenty": -1.0, "Squared distance to end": -0.5855680650590152, "Score": 1.339078154079141, "Brick": -0.47036192226189244, "Has largest army": 1.618222112287386, "Victory Point": 1.94, "Wool": 1.6035159107104948, "Num roads": -0.05521105644717017, "Ratio roads to

settlements": -4.9121348963244555, "Resource spread": 1.7125503304549476, "Has longest road": 1.2838670976319722, "Monopoly": -1.841636, "Devcards played": 0.6094224235774497, "Ore": -0.46214231042460535, "Longest Road": 1.2575471027049467, "Desert": -0.275532514214506632, "Ratio cities to settlements": -2.6105907106864046, "Num cities": 1.7313957055699865, "Knight": -1.8752206249592758, "Num turns with more than 7 cards": -1.1080078280902117, "Has Won": 1000.0, "Grain": -0.2244102767525466, "Num cards discarded": -1.7802727097158075, "Num accesible resources": 1.3717322013075097, "Road Building": -2.94, "Wood": -0.41156097255457064, "Num settlements": 0.0437112389534398}

Best Weight vector for basic Q-Learning AI:

{"Year of Plenty": 0.00010867371163066357, "Player 0 Roads": 0.02866592251993283, "Squared distance to end": -0.024362108563486678, "Score 1": 0.01443441626739214, "Score 0": 0.016604500918253735, "Score 3": 0.01160904595951565, "Wood": 0.0012941024686175947, "Player 3 Roads": 0.021719318507775653, "Brick": 0.001291597073765104, "Has largest army": 0.00011326389985333448, "Player 1 Settlements": 0.0033995408493797293, "Player 3 Settlements": 0.003055411213274853, "Victory Point": 0.00032782961349301374, "Wool": 0.001138646996813886, "Num roads": 0.05899578413810772, "Ratio roads to settlements": 0.02291518695870201, "Resource spread": 0.007039141619595944, "Has longest road": 0.0021252744829009 "Monopoly": 8.724036754413504e-05, "Devcards played": 0.001305987141503379, "Player 0 Settlements": 0.0036373271640526643, "Ore": 0.0012172965198637639, "Longest Road": 0.027315622951219393, "Player 1 DevCards": 0.0008562852704533374, "Player 1 Roads": 0.021904596827461213, "Ratio cities to settlements": 0.007129833510650611, "Num cities": 0.012221865039752176, "Player 1 Cities": 0.0045561093761149205, "Knight": 0.0008037208888830867, "Num turns with more than 7 cards": 0.03939138943748209, "Player 0 Cities": 0.00540961756850766, "Has Won": 8.435358282899886e-05, "Player 2 DevCards": 0.0006885715359185561, "Grain": 0.0014411478189401438, "Player 2 Roads": 0.024237464322012518, "offset": 0.0039872041590718055, "Num cards discarded": -0.008767733717452408, "Player 3 Cities": 0.0030207863143146465, "Player 0 DevCards": 0.001104733000184152, "Num accesible resources": 0.06568753453635601, "Road Building": 0.00010776497586961905, "Desert": 0.0, "Score": 0.0382575047740289, "Player 3 DevCards": 0.00044142935418607616, "Player 2 Cities": 0.00376156198948402, "Score 2": 0.013423244625034456, "Player 2 Settlements": 0.003503019935557245, "Num settlements": 0.004294386972394588}

Best weight vector for Q-Learning AI with win percentage as target:

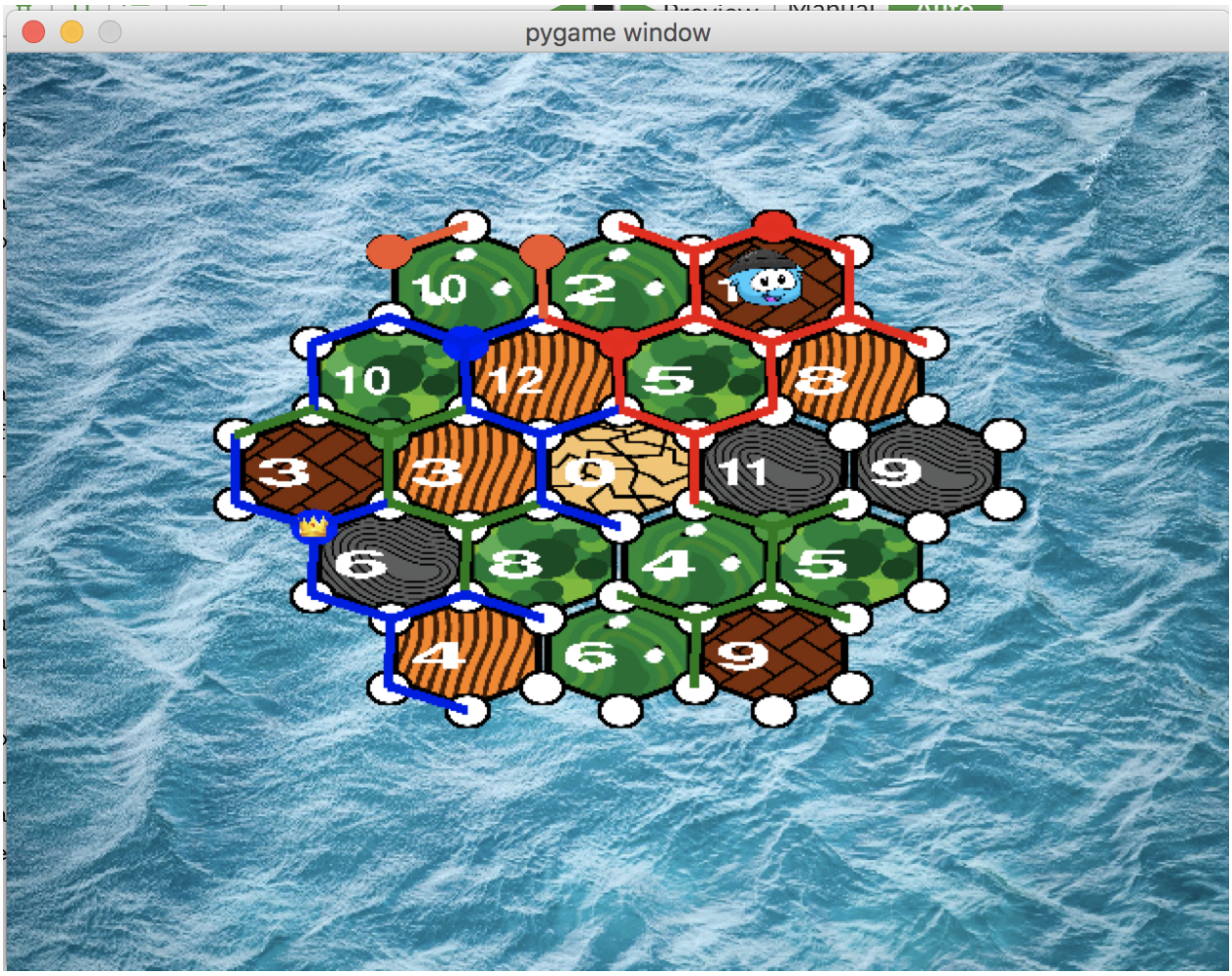
{"Year of Plenty": -1.2787765474653896e-05, "Player 0 Roads": 0.008931652870399386, "Squared distance to end": -0.024886658653861318, "Score 1": 0.006528913448164726, "Score 0": 0.006232667161924194, "Score 3": 0.006203571341557258, "Wood": 0.0008591854568805701, "Player 3 Roads": 0.008340280618455757, "Brick": 0.0008918699776369459, "Wool": 0.0008284694201915 "Player 1 Settlements": 0.002881642597245641, "Player 3 Settlements": 0.0020873336980664877, "Victory Point": 0.0001651392642439216, "Has largest army": 7.785481169995849e-05, "Num roads": 0.04003180857786877, "Ratio roads to settlements": 0.012508370634595666, "Resource spread": -0.004078344699959515, "Has longest road": 0.0015504455984081257, "Num cities": 0.0076604548472379306, "Devcards played": 0.00038633817248505647, "Player 0 Settlements": 0.0015938553597946252, "Ore": 0.0008418609519339178, "Longest Road":

0.016697395870372552, "Player 1 DevCards": 0.0002806493270544592, "Player 1 Roads": 0.008188007702806695, "Ratio cities to settlements": 0.004128911802702181, "Monopoly": 4.983940635529638e-05, "Player 1 Cities": 0.0012286426821129265, "Knight": 0.0002195129045768941, "Num turns with more than 7 cards": 0.0009579806350110832, "Player 0 Cities": 0.001853495637488969 3, "Has Won": 2.8121119999888326e-05, "Player 2 DevCards": 5.039296922404448e-05, "Grain": 0.0008234329987586492, "Player 2 Roads": 0.006978529955383161, "offset": 0.0025724720524281253, "Num cards discarded": -0.005863417606837882, "Player 3 Cities": 0.0017557140286496432, "Player 0 DevCards": 8.499269927753193e-05, "Num accesible resources": 0.04167819957338984, "Road Building": 6.265480535355721e-05, "Desert": 0.0, "Score": 0.026673351535729285, "Player 3 DevCards": 0.0004535657364619691, "Player 2 Cities": 0.0017263707656369337, "Score 2": 0.005992396350145855, "Player 2 Settlements": 0.0016590058845131004, "Num settlements": 0.003453630163887762}

Best weight vector for optimized Q-Learning:
{"Year of Plenty": 5.924978821441266e-06, "Player 0 Roads": 0.021605994625889267, "Squared distance to end": -0.025801401839007587, "Score": 0.03785961428873857, "Score 1": 0.012376831052358395, "Score 0": 0.012742056644287814, "Score 3": 0.012692185459272866, "Wood": 0.001325694070013062, "Player 3 Roads": 0.021069821215700956, "Brick": 0.0013222623725866103, "Has largest army": 8.586120855028905e-05, "Player 1 Settlements": 0.004037867516175987, "Player 3 Settlements": 0.0038472197276328197, "Victory Point": 0.0002034554379357742, "Wool": 0.0011475238706054318, "Num roads": 0.06513927688718141, "Ratio roads to settlements": 0.024398845715985405, "Resource spread": 0.004977877701420757, "Has longest road": 0.002291434804867, "Monopoly": 6.081537488970022e-05, "Devcards played": 0.0006371058475062538, "Ore": 0.001231573094916507, "Longest Road": 0.029713007799230553, "Player 1 DevCards": 0.000629583077127, "Player 1 Roads": 0.019750088281654685, "Ratio cities to settlements": 0.00622763779230201, "Num cities": 0.011333723128300617, "Player 1 Cities": 0.00291746370658629, "Knight": 0.00038707617920022376, "Num turns with more than 7 cards": 0.03770641246337631, "Player 0 Cities": 0.0036480078853040085, "Has Won": 2.9911404804468597e-05, "Player 2 DevCards": 0.0003450081261935012, "Grain": 0.0012130336580290742, "Player 2 Roads": 0.018424823816226874, "offset": 0.003910815656292461, "Num cards discarded": -0.008015575968064202, "Player 3 Cities": 0.00350760962775428, "Player 0 DevCards": 0.0002876812039642914, "Num accesible resources": 0.06356256670863115, "Road Building": 8.586071607940283e-05, "Desert": 0.0, "Player 0 Settlements": 0.003045218847887509, "Player 3 DevCards": 0.0005676279964055422, "Player 2 Cities": 0.0035575996404681814, "Score 2": 0.012098079370371771, "Player 2 Settlements": 0.0030571514358433923, "Num settlements": 0.0045636840604996775}

GitHub repository for this project: https://github.com/noahjax/CS221-Catan.git

Image of the final graphical display for the project




# Works Cited

[1] https://ilk.uvt.nl/icga/acg12/proceedings/Contribution100.pdf

[2]https://www.sbgames.org/sbgames2017/papers/COMPUTACAO/FULL_PAPERS/175405_2_versao_preliminar.pdf