Catan AI Player

Xander Minch, Liam Coveney, Sam Greenfield

May 6, 2025

1 Introduction

Catan is a strategy based board game in which players compete to earn 10 victory points (VPs) before the rest. The game functions the same every time, the board setup is different every game (over 20 billion permutations). The players earn points by using resources to build settlements, which yield resources when their corresponding number is rolled. They also build cities and roads between settlements and cities, or buy and play development cards. In addition, players may trade between themselves, trade via ports for better deals, and trade four of any resource for one of another with the bank. All of these mechanics are valuable and necessary at different points in the game, and it is crucial to have a strategy defining when to use them and how to value resources.
Because players are presented with such a large set of moves that usually grows as the game evolves (as opposed to chess, where there are fewer options as a player loses pieces), the implementation of an effective AI for Catan is computationally expensive. Using a Max^n algorithm (as in our project), searching 5 turns in depth can take multiple seconds. So we were forced to figure out a work around as well as optimize the system to make the AI competent within a reasonable run time.

Furthermore, there is no reasonably simple strategy to guarantee a win or winning position. As the board changes with every game, a player's strategy must change with it. Generally, a player would like to play on spaces which both diversify the resources they can receive and maximize the likelihood of receiving a resource given a dice roll. Simply put, a player may attempt to build a settlement on every resource, with those being on hexes with numbers that are likely to be rolled. However, because initial settlements are chosen via a snake draft, it is sometimes impossible to even guarantee an avenue of income for every resource.

2 Algorithms

2.1 Stochastic Max^n Algorithm

We implemented the Max^n algorithm, an extension of the traditional minimax approach designed for games with more than two players. In minimax, the algorithm alternates between maximizing and minimizing layers at each depth of the game tree: the current player seeks to maximize their own advantage, while the opponent attempts to minimize it on their turn. In contrast, Max^n assumes that each of the n players is independently trying to maximize their own heuristic value. As in 1, the game tree takes into account turns between players rather than switching between a min and a max player. At each layer of the game tree, there is a list of heuristic values corresponding to the strength of each player.

To account for the stochastic nature of Catan, which comes in the form of random dice rolls at the beginning of each turn, we incorporated chance nodes into our tree. These nodes simulate all possible dice outcomes weighted by their probabilities, allowing the AI agent to compute expected heuristic values. In our implementation, the Max^n algorithm chooses the move which maximizes the differential between the current player's heuristic value and the average heuristic of the other players. As evidenced in 1, the game tree can become large, making computation expensive. The branching factor is variable since the number of valid moves a player can make is based on the player's position. Thus, when calling the algorithm, we increment the depth of Max^n until our set time limit is reached to guarantee a move. When the player has no resources or development cards, only one option (end turn) is available. When the player has many cards, they can make various moves. We estimate that

on average, a player has around 10 options of moves to make.

The time complexity of this algorithm is similar to that of a regular minimax algorithm, and is based on the number of moves a player can make, number of dice outcomes, and depth. Minimax has time complexity $O(b^d)$, where b is the branching factor and d is the depth of search. With the increased branching due to dice rolls, our algorithm has a time complexity of $O((c * b)^d)$, where c is the number of possible dice rolls. All this in consideration, based on $c = 11$ for each possible dice roll, an estimated average branching factor around $b = 10$ moves per turn, and depth d from iterative deepening, the time complexity of our Max^n is approximately $O(110^d)$.

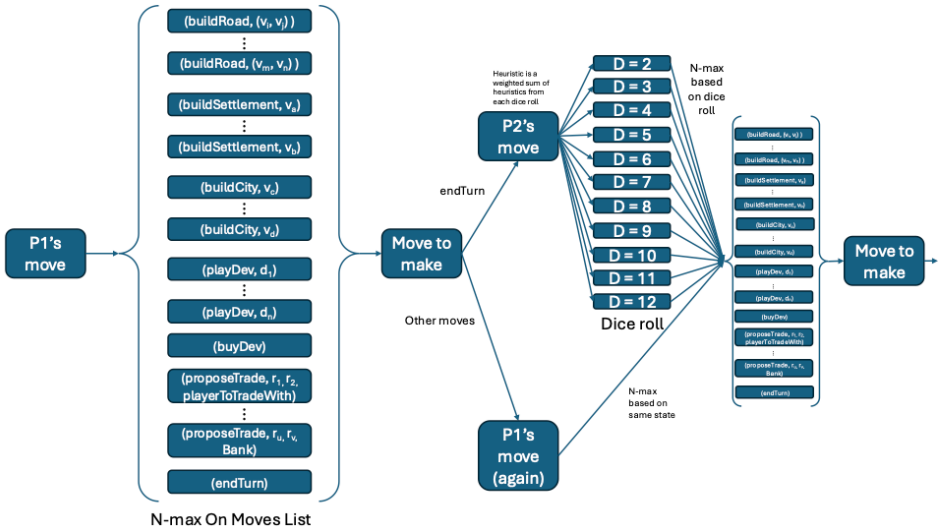

Figure 1: Stochastic Game Tree Visual of Model

2.2 Heuristic

Our heuristic evaluates four aspects of the game state: board positioning, victory point value, building value and development card value. These values combined into a singular dot product with their weighted value of importance we determined through trial and error. When evaluating board positioning, the method encourages having a settlement or city on every resource, encouraging the AI to work toward spots that include resources it has not settled yet.

We also factor in the probabilistic weighted value of the player's settlements on the board, by taking the dot product of the AI's resource distribution multiplied by those tiles' numbers likeliness to roll, promoting the strategy discussed in the introduction. This is weighted with little value (.5) as we found anything more leads the player away from winning moves in end game.

The method for the VP value of the player is fairly simple. It first calculates the "visibleVP" of the other players. This is needed because there are development cards which give players VPs and are hidden from the other players in the game, and there are visible VPs that players gain through settlements (1 VP), cities (2 VPs), longest road (2 VPs), and largest army (2 VPs). For each player, we sum these values and include a "likelihood that a development card is a VP" factor, then compare the AI's VPs to the game average as well as giving extra credit to the player with the most points. This is weighted the highest among values considered (3), as choosing a play that increases a player's VP is generally the best move.

Development card value simply adds value if the player has largest army or longest road, as well as small additional points if one of the resource-garnering cards is in hand. This value is weighted middle of the road (1), since those values discussed have high-yield VP.

Lastly, the building value is cumulatively weighted with 8 points among its helper functions as we wanted to strongly encourage our AI to build structures. We found this was the best approach since in previous iterations, our AI would lose games because it was to eager to hold onto its resources.

2.3 Initial Settlement Algorithm

Our Max^n algorithm did not translate well to the beginning settlement rounds because the amount of options a player gets on one of their first turns is too great to accurately find an optimal move within the created game tree. With our initial settlement logic, players would choose spaces at random, which led to little success in the main game because of how poor their setup was. We implemented simple logic to select the space that maximizes the probability of getting a resource every round. This is one of a few main strategies that human players use for this phase of the game, and using it every time gives our AI agent a good enough setup to regularly find a successful structure during the main rounds of the game.

2.4 Limitations and Alternatives

Catan is a computationally expensive game, with tons of possible moves per turn. In the sense of the Max^n, since the branching factor correlates to possible legal moves, we had to adjust our player to not consider moves too far in depth so as to account for the real pace of play for a Catan game. Thus, we limit the amount of time the program can take to consider moves.

Some alternatives to our method of implementation could have been a machine learning model that is trained on game data, using something like reinforcement learning to determine the best move. While this method is intriguing, our team had little experience with machine learning, and simply found the heuristic approach more appropriate and timely for the project. In addition, Catan is high-dimensional in its move set, and we determined that a machine learning approach could lead to over-fitting models, which is suboptimal for new game decisions.

3 Human Thought Processes

Our heuristic tells our model what is valuable and what is not, so while creating it, we had to think about how we and others played the game. We valued resources and moves based on different strategies that human players use. For example, some people pursue the longest road card and prioritize road-building more than a player that is trying to win from development cards. Our heuristic checks which strategy it is most in line with, and assigns weights based on that.

For the initial two rounds of the game, the placement rounds, the game mechanics work differently. During these rounds, the goal of the AI is to place settlements which optimize resource production. Because of this one-dimensional optimization, the AI agent works under the assumption that it will be able to build toward or trade for resources it does not initially settle on.

4 Findings

4.1 Performance

Certain moves require certain resources, so building a settlement or even a road is made extremely difficult if a player has no income of a needed resource. This makes initially choosing where to place settlements a hugely important part of the game. In the case of the random player our team tested the AI against, we found the random player often struggled due to the failure of its beginning random choice. Therefore, our AI agent started the game with a huge advantage.

So, our performance evaluation is split into two sections: one where the random player chooses initial settlements randomly, and one where it uses the same algorithm our AI uses. We find that while the results after giving the random player help in the beginning are less impressive, they do better highlight areas where our project could be improved in the future.

4.1.1 Max^n vs. Total Random

In testing, we ran 50 games against two totally random agents. We randomized the order of players each time to test different situations for our agent. We set the time limit for our AI to 0.2 seconds so as to simulate the games faster. Even in that time, the depth still reached around 5 turns in search.

In those 50 games, our AI won 47 times for a total win rate of 94%. The two random players won a combined 3 times. To demonstrate more clearly how handily the AI player was winning, our

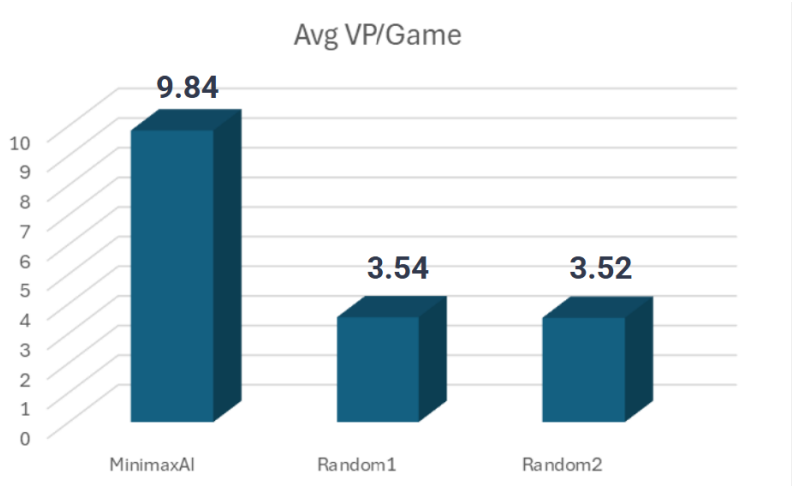

Figure 2: VP Averages for 3 Agents over 50 games

team recorded the VP count of each player at the conclusion of each game. On average, the AI player reached 9.84 VPs/game. The smallest number of VPs at the end of a game our AI recorded was 5, demonstrating competence even when the agent lost.

The random players averaged 3.53 VPs. This means that they, in general, only placed for around 1.5 VPs after initial settlement placement (the two settlements placed before normal game play each count for one VP). This shows not only how hard Catan is, but also the importance of beginning placement.

4.1.2 Max^n vs. Initially-Competent Random

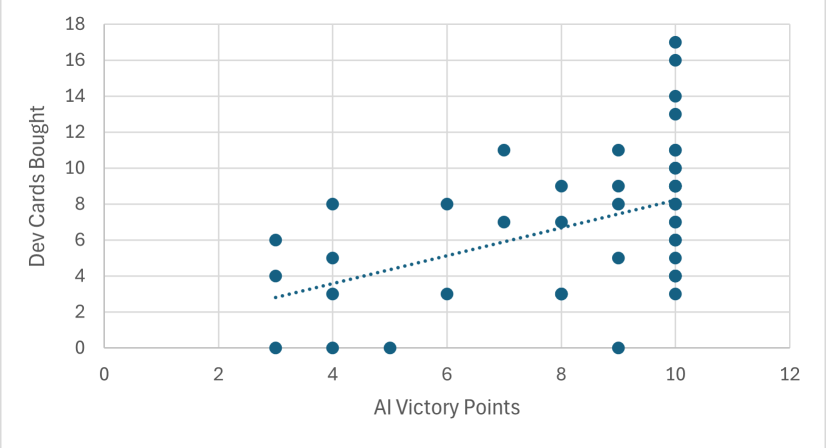

Figure 3: Victory Points vs Development Cards Bought by Max^n Agent

In these tests, similarly, 50 games were ran with a 0.2-second time limit against two random AI agents, but the random agents were allowed to pick settlements for optimal resource generation. In these 50 games, results were better distributed. Our AI won 56% of the time, which is significantly above the expected 33% for a 3 player game. Our AI averaged 8.42 VP per game, with the improved random players at 6.22 and 6.6 this time around. This differential shows the performance of our AI is deeper than just its win rate, that even with the randomness of dice roll our agent is still consistently close to winning.

Because initial setup is so important, and because the resource production of a player's initial settlements is determined by player order, we noted in these tests the win rate by placement of our agent in the snake draft: When going first, we had a 52% win rate, second had 50%, and third had

67%.

In these tests, we also analyzed gameplay tendencies. On average, our AI built 4.24 Settlements/game, and 0.28 Cities. In making the agent, we found it difficult to urge the AI to build cities even though their construction can be of great importance. In our tests, we found when 1 or more cities are built, the AI won 73% of the time.

On average, our AI bought 7 development cards. Strategies prioritizing buying development cards are widely popular among Catan players. As shown in 3, the purchasing of development cards is positively correlated with VP counts in end game. The agent only won games where it bought at least three development cards, and averaged higher scores when it bought more. This suggests that development card investment is an effective strategy across varied game conditions.

We additionally tracked whether the player obtained largest army or longest road. In total, largest army was obtained 20 times and longest road 37. How these achievements affected the win rate is shown in the following table:

	With Army	**Without Army**
With Road	100%	40%
Without Road	67%	10%

Table 1: Win Rate by Longest Road or Largest Army Status

These results suggest a strong correlation between having largest army and/or longest road and winning the game. When the AI secured both, it had a perfect win rate (1.0), while having just one of the two still significantly increased its odds of winning. In contrast, when the AI had neither, the win rate dropped sharply to just 10%, highlighting the strategic importance of these achievements.

Overall, these findings demonstrate that our AI agent is capable of consistently outperforming competent opponents through strategic prioritization of development cards and key achievements. Its performance highlights the effectiveness of our heuristics and suggests that further gains may come from improving city construction behavior and optimizing trade strategies.

4.2 Comparison with Alternate Implementation

We will compare the results of our bot to a few different agents from this Medium article, which describes a similar project. The first of the bots is the ValueFunctionPlayer. This implementation is fairly similar to our heuristic/Max^n approach. The benchmarks, or other players, used to test against this bot are RandomPlayer and WeightedRandomPlayer, both fairly similar to our two versions of random player. In 1000 tests of a 3 player game, their value player won 99.1% of games, whereas ours won 94%.

5 Outside Code

We used GitHub user kvombatkere's Catan-AI repository (https://github.com/kvombatkere/Catan-AI) as a basis for this project. The project has an MIT License, which allows for us to build off of the project. The original license is included in our source code, and a full report of additions, removals, and changes to the original project is available in our README.

The model uses pygame to make a visual representation of the game, and tracks game statistics on the backend to determine winners and game resources. We significantly altered the code and game logic to suit our project and the rules of Catan. The code we used had flaws in the game logic for fundamental parts of the game, such as resource and development card allocation and trading. We fixed these problems and were able to adapt it to our purposes, implementing trading and ensuring the game progresses as it should.

We also implemented a class that allowed us to save the game state from the adapted code as an object called gameState, and used this as the core of our Max^n algorithm. Using this, we were able to save and games, and evaluate how well players are doing given a gameState. We also added a tracker for each of the players' information so we can monitor how the game is running and how and why decisions are made. This more robust version of the game gave us a foundation to build and test our model.

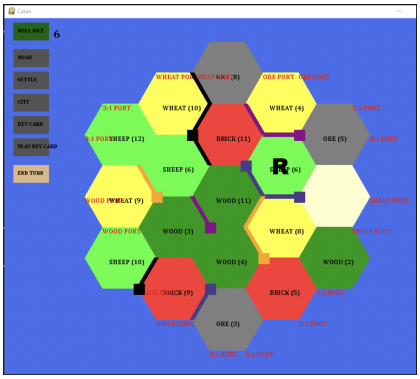

Figure 4: The model when we cloned it from GitHub.

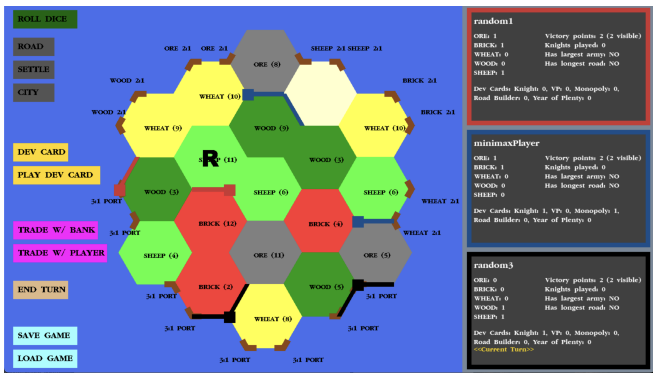

Figure 5: The model after our edits and significant fixes for game logic.

6 How Our Model Could Improve

Given more time we would train the AI more to make more accurate decisions. We might consider using techniques more based around machine learning for different strategies in game. While we felt that a total ML approach wouldn't be appropriate, possible implementation for initial settlements or trade might have improved our performance.

As well, we could have made a more in depth algorithm for trade. Currently we just identify a resource we need and then add a singular trade to the possible moves. However, it is feasible to suggest we could have evaluated a larger list of trades, or had a model predict when a trade is most necessary. Our model also only accepts trades based on trying to even out its resources. In future iterations, we could allow the algorithm to use Max^n to determine whether a trade is good.

Additionally, our logic for the initial setup rounds currently selects a space based on maximizing probability of collecting a resource each round. While this is a viable strategy that humans use, there are other strategies such as trying to monopolize a resource or getting a port to facilitate easy trade which may be more successful depending on the board. To improve our model we could find a way to analyze the board, determine which strategy it is most conducive to, and move forward with that strategy in mind. This would make our agent more directed and intentional than it currently is.