Catan Conqueror: Building Bots to Rule the Island

Caroline Cahilly, Charlie Gordon, Proud Mplala
Stanford University
ccahilly@stanford.edu, charlieg@stanford.edu, pmpala@stanford.edu

Abstract

This paper explores the development and evaluation of intelligent agents for a two-player version of the board game, Settlers of Catan. In our study we implemented five distinct agents: a random agent, a custom value function agent, a Q-learning agent, a lookahead-with-rollouts agent using a random policy, and a lookahead-with-rollouts agent using a greedy policy (based on our custom value function). In our analysis we evaluated the performance of our agents through extensive agent-vs-agent simulations, comparing the win rates and average victory points earned. After significant training (15,000 iterations) of our Q-learning agent, it proved to be the most successful agent achieving an 85% win rate against its trainer, the custom value function, which was the second best performing agent. We also conducted an ablation study on the lookahead-with-rollouts agent, exploring rollout depths from 1 to 10, which revealed that deeper rollouts do not significantly improve performance beyond a depth of 5 but do increase computational time. Our results provide insights into the effectiveness of different AI approaches in the complex decision space of Catan and suggest potential areas for future work, such as further enhancement of the game engine and potentially better opponent modeling.

1 Introduction

Our goal is to develop intelligent agents that can play a two-player version of the board game "Settlers of Catan." Catan is a strategic board game where players compete to build settlements, cities, and roads on a modular hexagonal board representing an island rich in resources like lumber, brick, wool, grain, and ore. Players gather resources based on dice rolls and trade with others to develop their infrastructure and earn victory points, aiming to reach a total of 10 points first.

Our game works as follows. There are two agents, Agent 0 and Agent 1. For simplicity, we initialize the board (which consists of the hexagons, number tiles, and robber) the same way for each game (Figure 1a). The bank starts with 19 of each type of resource: brick, wool, ore, grain, and lumber.

(a) Initial Board (b) Example Initial Settlement/Road Placements

Figure 1: Example start of game.

Then, the players choose their initial settlements and roads (Figure 1b). Agent 0 (in red) places a settlement and a road. Agent 1 (in blue) places two settlements and two roads. Lastly, Agent 0 places a settlement and a road. Then, the agents collect their initial resources: they get one resource for each hexagon adjacent to the second settlement they place.

After initialization, each turn proceeds as follows: The player whose turn it is rolls two dice, producing a result between 2 and 12. Any hex with a number (except 7) matching the roll produces resources (unless the robber is on it, in which case it does not produce resources). Players with settlements or cities adjacent to such hexes collect resources from them: one resource per adjacent settlement and two resources per adjacent city. In the event that the players are supposed to collect more resources than are in the bank, they collect none.

If a 7 is rolled, then any player who has more than seven resources must get rid of half of their resources. Next, the agent whose turn it is must move the robber; if the other player has a settlement/city adjacent to this robber hex, the current player gets to steal one random resource from them.

Then, the player can take as many of the following actions, assuming they are valid, until they choose to pass:

- **Build a road.** Agent must give one lumber and one brick to the bank. They can build on any undeveloped edge adjacent to one of their existing roads, settlements, or cities. Each player can build up to 15 roads.

- **Build a settlement.** Agent must give one lumber, brick, grain, and wool to the bank. They can build on any vertex that is at least two edges away from any other settlement/city and adjacent to one of their roads. Each player can build up to 5 settlements.

- **Build a city.** Agent must give two grain and three ore to the bank. They can build a city to replace any of their existing settlements. Each player can build up to 4 settlements.

- **Draw a development card.** Agent must give one grain, wool, and ore to the bank. There are 25 cards, of five different types: knight (14), victory point (5), road building (2), year of plenty (2), and monopoly (2). Drawing the "Victory Point" card automatically gives the player one victory point. All other cards must be played in a subsequent turn.

- **Play a development card.** Agent must have a development card in their deck, and then discard the dev card after playing it. They can only play one dev card on a given turn. The cards work as follows:
  - **Knight.** Agent moves the robber to a new hex and steals a resource card from the other player if they have a settlement/city adjacent to that hex.
  - **Road building.** Agent builds two roads, assuming that there are free spots and they have not already built 15 roads.
  - **Year of plenty.** Agent chooses one resource and takes all of that resource from the other player.
  - **Monopoly.** Agent chooses two resources to take from the bank.

- **Trade.** Trade with the bank to get one resource in exchange for four cards of a single resource type.

- **Pass.** Take no action; the next player starts their turn.

The victory points are assigned as follows. Players get:

- One point for each settlement they have
- Two points for each city they have
- One point for each victory card they draw
- Two points if they have the "Longest Road" card. To claim "Longest Road", the player must have a continuous road of at least 5 segments. The other player can steal the card if they build a longer road. The player can potentially lose this card if the other player builds a settlement in the middle of their road.

- Two points if they have the "Largest Army" card. To claim "Largest Army", the player must have played at least 3 knight dev cards. The other player can steal the card if they play more knight dev cards.

The first player to 10 victory points wins.

We see that there is a large action space for the game, and a large state space defined by the board state and the resources and dev cards possessed by each player. Additionally, each player faces uncertainty introduced by the dice rolls, their opponent's moves, their opponent's dev cards, the dev cards they draw, and what card gets stolen (if any) when the robber is moved.

In our project, we create four agents that play this game: (i) a random agent, (ii) a custom value function agent, (iii) a Q-learning agent, (iv) a lookahead-with-rollouts agent. We compare each bot's performance to two benchmarks: (i) random-move bot and (ii) average human-level competition. We also evaluate how the agents perform against each other.

2 Related Work

Many people have attempted to create bots to play various versions of Catan. We used the game simulator by (Kaplan-Nelson et al., 2014) as our starting point. Kaplan-Nelson et al. (2014) created random and minimax agents that could play Catan in a three-player setting. Their game was simplified, however, as it did not include the development cards and associated actions, the robber piece and associated actions, the possibility to trade with the bank, or a "human player" option, which we added. They found that their agent won about 80% of the time when playing two random agents.

We also used work by Collazo (2024) as a starting point for our custom value function used by our custom value function and lookahead-with-rollouts players. Their value function evaluated a current state's utility by weighting intermediate rewards, such as victory points, amount of production (expected resource collection after a dice roll), opponent production, number of vertices where new settlements could be built, etc. (see section 3.2.1 for more details). We then adapted the weights to better fit our specific state space. For example, we adjusted the weight for victory points for numerical stability reasons. We also added a variable for winning or losing the game, which helped in particular when evaluating the value of a given rollout in our lookahead-with-rollouts players.

3 Methods

We implemented four agents: a random agent, a custom value function agent, a Q-learning agent, and a lookahead-with-rollouts agent.

3.1 Random Agent

In the initialization phase, the agent places their first two settlements and roads totally randomly. Then, for each turn, the agent keeps choosing an action from their set of valid actions until they choose to pass, when it becomes the next player's turn.

3.2 Custom Value Function Agent

For game initialization, this agent picked its initial roads and settlements by evaluating the quality of settlement spots using a custom function that took into account production, the variety of resources produced, and the ability to build toward optimal new spots to settle. Then, for all subsequent turns, we implemented a greedy policy that (i) takes the list of all possible actions, (ii) evaluates the utility of taking each action with the custom value function described in section 3.2.1, (iii) takes the action with the highest utility, and (iv) repeats until the chosen action is "pass".

3.2.1 Custom Value Function

Our custom value function agent takes into account the following variables. They are listed from highest to lowest weight. Note that weights can be negative. We chose the weights empirically, using work by Collazo (2024) as a starting point. We weighted 15 features. The top three positive-weighted features were (i) winning, (ii) number of victory points, and (iii) reachable production, meaning the

production if the player were to extend each road by one and then settle these vertices. The top three negative weighted features were (i) losing, (ii) opponent production, and (iii) being forced to discard cards when a seven is rolled.

3.3 Q-learning Agent

The Q-learning agent uses a model-free reinforcement learning approach to learn an optimal policy for playing Catan. This agent learns by interacting with the environment and updating its Q-values based on the rewards it receives. To handle the complexity of Catan, this implementation incorporates several advanced techniques beyond basic Q-learning.

3.3.1 State Representation

The state is represented as a comprehensive tuple encompassing key game elements: the agent's victory points, settlement count, city count, road count, and resources in hand; the opponent's victory points; the robber's position; possession of longest road and largest army; development card count; total bank resources; both players' longest road lengths; and the current game stage (early, mid, or late). This detailed representation captures the game's complexity, allowing the agent to make informed decisions based on its progress, the opponent's status, and the overall game situation.

This comprehensive state representation allows the agent to capture important aspects of the game state, including its own progress, the opponent's progress, and the overall game situation. The inclusion of the game stage helps the agent adapt its strategy as the game progresses.

3.3.2 Q-learning Algorithm

The agent uses the Q-learning algorithm to update its Q-values:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')] \tag{1}$$

where s is the current state, a is the action taken, r is the reward received, s is the next state, is the learning rate, and γ is the discount factor.

3.3.3 Exploration Strategy

The agent uses an ϵ-greedy strategy for exploration:

$$\epsilon = \max(\epsilon_{\text{end}}, \epsilon_{\text{start}} \cdot \epsilon_{\text{decay}}^{t}) \tag{2}$$

where ϵ_{start} is the initial exploration rate, ϵ_{end} is the minimum exploration rate, ϵ_{decay} is the decay factor, and t is the number of iterations.

To improve learning over time, the learning rate α decays according to:

$$\alpha = \max(\alpha_{\text{min}}, \alpha_{\text{initial}} \cdot \alpha_{\text{decay}}^{t}) \tag{3}$$

where α initial is the initial learning rate, α_{min} is the minimum learning rate, α_{decay} is the decay factor, and t is the number of iterations.

3.3.4 Experience Replay

To improve learning stability and efficiency, the agent uses prioritized experience replay. Experiences are stored in a buffer and sampled based on their TD-error, allowing the agent to learn more effectively from important experiences.

3.3.5 Reward Shaping

The reward function is designed to encourage actions that lead to victory:

- +100 for increasing victory points

- +50 for building a settlement
- +75 for building a city
- +25 for building a road
- +200 for getting the longest road or largest army
- +1000 for winning the game
- -1000 for losing the game

Additional rewards are given based on the game stage and the specific actions taken, such as bonuses for building towards good settlement spots or penalties for unnecessary roads. This sophisticated reward shaping helps guide the agent towards effective long-term strategies.

3.3.6 Opponent Modeling

To manage memory usage and focus learning on relevant states, the agent periodically prunes rarely visited states from its Q-table. This is done by removing states that have been visited less than a certain threshold number of times. This technique helps to prevent the Q-table from growing unnecessarily large and allows the agent to concentrate on the most important parts of the state space.

3.3.7 Q-table Pruning

To manage memory usage, the agent periodically prunes rarely visited states from its Q-table, focusing on the most relevant parts of the state space.

3.4 Lookahead-With-Rollouts Agents

We implemented lookahead-with-rollouts agents of varying depths with either a random policy or a greedy policy with respect to the custom value function from section 3.2.1 depending on the experiment. For the random rollout policy, each action is picked randomly. For the greedy rollout policy, each action is picked based on which action maximizes the same custom value function.

For initialization, both versions of this agent picked its first two settlements and roads in the same way as the custom value function agent (section 3.2).

Then, for each turn, they selected actions based on the policy until the action selected was "pass". The opponent's actions were simulated with a random or greedy policy depending on the experiment. We rolled out to a depth of d depending on the experiment, meaning that we simulated d turns by the player and d turns by its opponent. The value of each rollout was evaluated as the sum of the value of each state reached on the rollout (discount factor of 1). We calculated these values with the custom value function (section 3.2.1).

4 Experiments

4.1 Agent vs. Agent

For each pair of the following four distinct agent types (random, custom value function, Q-learning, lookahead), we made them play against each other 200 times, with each agent going first 100 times. The lookahead agent had depth $d = 10$ and used a random policy for both its and its opponent's actions. Additionally, for each agent type, we made it play against itself 100 times. We recorded their win rates and the average victory points of both players at the end of all games.

4.2 Ablation Study for Lookahead-with-Rollouts Agent

We set the depth for the lookahead agent to $d = 1, 2, 3, 5, 10$ and used a random policy for both the agent and its opponent. Then, we had it play against a random agent. We ran 100 game simulations for each depth and investigated how the depth affected the win rate, average victory points, and time per game.

4.3 Policy Choice for Lookahead-with-Rollouts Agent

We set the depth for the lookahead agent to $d = 2$. We then tried the following rollout policies (i) random policy for both agent and opponent, (ii) greedy policy for agent and random policy for

opponent, and (iii) greedy policy for agent and greedy policy for opponent. Then, we had it play against the random agent. We ran 100 game simulations for each policy combination and investigated how policy choice for self and opponent affected win rate, average points, and time per game.

5 Results & Discussion

For both experiments, we reported the win rates and average victory points at the end of the game for each pair of players. We recorded win rate because the ultimate goal of the agent is to win the game. We recorded average victory points at the end of the game to understand how close each game was.

5.1 Agent vs. Agent

	Agent 1			
Agent 0	Random	Custom Val. Func.	Q-learning	Random Lookahead
Random	51	0	0	3
Custom Val. Func.	100	86	33	92
Q-learning	100	85	77	89
Random Lookahead	97	33	17	78

Table 1: Win Rate of Agent 0 (%)

As seen in Table 1, the Q-learning agent was able to perform the best against the other agents. In our initial data collection we found that the Custom Value Function, was originally our best performing agent, so we decided to train the Q-learning agent against it for 15000 iterations both as player 0, and player 1. This training proved to be very effective improving our initial win rates of around 25/30% to the 85% we ended up with.

The Random agent, as expected, was our worst performing agent. Though it was very useful in providing a base on which to build the other agents up from, and develop the game engine.

The Random Lookahead function was very consistent at beating the Random agent, but not so much against the other agents. This is likely due to our random rollout policy being suboptimal. If we were to work further on this, we would probably base the lookahead function on what the actions of the custom value function or q learning agents would be.

As you can see from the diagonal in the table, the agents typically had an advantage when going first, except for the Random agent. This is likely due to the initial settlement spots on the board, with there being a pretty significant advantage of taking the best spot available on the board. This is why we tested the agents from both player 0 and player 1, in order to eliminate any bias.

	Agent 1			
Agent 0	Random	Custom Val. Func.	Q-learning	Random Lookahead
Random	7.1 / 7.5	10.1 / 3.0	10.2 / 2.6	10.2 / 3.2
Custom Val. Func.	2.9 / 10.2	6.1 / 9.7	9.0 / 6.8	6.2 / 9.9
Q-learning	2.7 / 10.1	6.5 / 9.5	6.0 / 9.1	5.7 / 9.8
Random Lookahead	3.2 / 10.2	9.2 / 6.8	9.6 / 6.1	7.0 / 9.6

Table 2: Average Victory Points of Agent 0 \ Average Victory Points of Agent 1.

In Table 2, we observe that for all pairs that do not include the random player, the loser achieves higher average victory points when it goes first than when it goes second. By contrast, when the random player loses, the victory points do not vary much depending on if the random player it goes first or second. We also see that when a bot plays itself, the first player achieves higher average points than the second player. Again, this shows that the ability to go first when choosing initial settlements and roads strategically improves game performance.

5.2 Ablation Study for Lookahead-with-Rollouts Agent

Table 3: Ablation Studies for LookAheadRolloutPlayer

Rollout Depth	Win Rate (%)	Average Victory Points	Time per Game (s)
1	99	10.18	1.85
2	99	10.28	2.63
3	99	10.21	3.37
5	100	10.16	5.41
10	99	10.28	9.27

While the win rate remains consistently high, our results show that deeper rollouts do not significantly improve scoring consistency. However, deeper rollouts incur a noticeable computational cost, with the time per game increasing from 1.85 seconds at depth 1 to 9.27 seconds at depth 10, underscoring the trade-off between decision-making complexity and efficiency.

5.3 Policy Choice for Lookahead-with-Rollouts Agent

Table 4: Performance Metrics for LookAheadRolloutPlayer (Depth=2)

Rollout Policy	Win Rate (%)	Average Victory Points	Time per Game (s)
Random Policy, Random Policy	97	10.21	3.15
Greedy Policy, Random Opponent	98	10.30	3.23
Greedy Policy, Greedy Opponent	100	10.18	2.87

For the rollout policy, we see a steady increase in the win rate from random policy to modeling agents in a greedy manner. We imagine this is because modelling an opponent as a greedy accounts for the worst case and thus leads to better outcome.

6 Conclusion & Future Work

In this study, we developed and evaluated four distinct agents for playing a two-player version of Settlers of Catan: a random agent, a custom value function agent, a Q-learning agent, and a lookahead-with-rollouts agent. Our experiments revealed that the Q-learning agent, after extensive training against the custom value function agent, emerged as the strongest performer, achieving an 85% win rate against its trainer. The custom value function agent proved to be the second-best performer, consistently outperforming the random and lookahead agents.

Our ablation study on the lookahead-with-rollouts agent showed that increasing the rollout depth beyond 5 did not significantly improve performance but did increase computational time. The study also revealed that a greedy policy for both the agent and simulated opponent actions yielded the best results in terms of win rate and efficiency.

These findings demonstrate the potential of reinforcement learning techniques in complex game environments like Catan. They also highlight the importance of balancing computational cost with performance gains when implementing lookahead strategies.

Future work could focus on enhancing the game engine to include randomly initialized boards, ports, and player trading. Expanding the game to a 4-player setting would also provide opportunities to explore multi-agent dynamics and potentially more sophisticated opponent modeling techniques.

References

B. Collazo. 2024. Catanatron: An ai for settlers of catan. `https://github.com/bcollazo/catanatron`. Accessed: 2024-12-09.

S. Kaplan-Nelson, S. Leung, and N. Troccoli. 2014. An ai agent for settlers of catan. `https://github.com/skleung/cs221/blob/master/progress.pdf`. Accessed: 2024-12-09.