Computational Intelligence for
Settlers of Catan

Author
Omar Ashraf

Supervisor
James Glenn

Partner
Christopher Kim

December 12, 2018
Contents

1 Abstract ......................................................... 3

2 Introduction .................................................. 3

3 Simplifications ................................................ 5

4 Game Framework ............................................... 6
   4.1 Overview ..................................................... 6
   4.1.1 Game Board ............................................... 6
   4.1.2 Game Logic ............................................... 6
   4.1.3 Player .................................................... 8
   4.2 Performance Benchmarking ................................. 8

5 User Interface ................................................ 9

6 Intelligent Agents ............................................ 10
   6.1 Existing work ............................................... 10
   6.2 Random Agent .............................................. 11
   6.3 Heuristic Agents ........................................... 11
      6.3.1 Selected Heuristics .................................. 11
      6.3.2 Naive Heuristic Agent ................................. 12
      6.3.3 Smart Heuristic Agent ................................. 13
      6.3.4 Expert Heuristic Agent ................................. 13
   6.4 Q Learning .................................................. 13
      6.4.1 Overview ............................................... 13
      6.4.2 Linear Approximation ................................ 13
      6.4.3 Agent Design ........................................... 14
   6.5 Results ..................................................... 16

7 Future Work .................................................. 18
8 Distribution of Work
Abstract

Games serve as a useful domain to experiment with computational intelligence techniques. In this collaborative work, we investigate the use of linear approximation-based Q learning to develop a sophisticated agent that can competitively play the game, Settlers of Catan. We identify the potential of reinforcement learning strategies in developing a competitive player based on prior investigations into the applicability of learning-based methods to Catan. Pfeiffer employed a hierarchical reinforcement learning approach involving the use of model trees to learn high-level and low-level policies while Szita et al. incorporated domain-specific knowledge with domain-agnostic Monte-Carlo Tree Search to develop a competitive player. We pursue a similar but distinct approach that combines Q-learning function approximation with a priori heuristic-based knowledge to develop a agent that can rival a human’s performance. We start by creating a Python-based game framework that allows human players and artificial intelligence agents to play the game. The framework is designed to be decoupled from the user interface to run automated simulations and benchmark our agents’ performance. We implement several agents employing heuristic and learning-based strategies and evaluate their performance based on win percentages averaged over 5000 trials. Our results demonstrate the success of the linear-approximation-based Q learning agent that learns meta-actions mapped via heuristic-based knowledge to primitive game moves; this learning agent consistently defeats all other heuristic-based agents by a significant margin, and never falls below a 38.5% win rate. Future work will build upon this result by improving the heuristics used and learning high-level strategies to complement the meta-actions learned.

Introduction

This work seeks to address two main goals: 1) develop a functional Settlers of Catan (henceforth referenced to as Catan) program using the Pygame library that allows humans players to play against other human players and artificial intelligence agents, and 2) apply com-
putational intelligence concepts to design and implement an artificial intelligence agent to competitively play Catan.

Settlers of Catan is a stochastic board game played between 2-4 players on a board of hexagonal tiles. Each tile is assigned a particular resource (brick, wood, sheep, wheat, or ore) as well as a numeric value between 2 and 12 (inclusive). The configuration of these tiles is randomly generated and subject to several constraints to ensure fairness. The goal of the game is to accumulate 10 victory points which can be earned in the following ways:

1. **Settlements** - Settlements are placed on the corners of hexagonal tiles and constitute the primary means of acquiring resources. On each player’s turn, two dice are rolled and if a player has a settlement adjacent to a tile that contains the number that was rolled, he or she receives a resource card that matches the resource of the tile. Players place two settlements in the initial stage of the game and can build up to 5 settlements by trading in specific resource cards. Each settlement counts as one victory point.

2. **Cities** - Cities are upgraded settlements and can be built by trading in specific resource cards. If a player has a city adjacent to a tile that contains the number that was rolled, he or she receives two resource cards of the corresponding resource. Each city counts as two victory points.

3. **Longest Road** - Settlements and cities are connected to one another by roads. If a player builds 5 connecting roads, he or she attains Longest Road which corresponds to two victory points. Unlike settlements and cities, the owner of Longest Road can change throughout the game as players build additional roads.

4. **Largest Army** - Resource cards can also be used to buy development cards. There are 3 types of development cards: ability cards, which allow players to use one of three abilities; knight cards; and victory point cards. Knight cards allow players to steal resources from other players. When a player activates 3 knight cards, he or she attains Largest Army, which corresponds to two victory points. Just like Longest Road, the owner of Largest Army can change throughout the game as players activate additional knight cards.
5. **Victory Point Card** - The final category of development cards is the victory point card. When a player plays a victory point card, he or she is awarded a single victory point.

The game is divided into two phases: an initial *snake* phase during which players place one settlement and one road in one specific turn sequence, and a second settlement and a second road in the opposite sequence. This phase is succeeded by the turn-based gameplay where each player rolls the dice to start their turn.

**Simplifications**

We made two major simplifications to the traditional gameplay for the sake of ease in learning. The first simplification was to eliminate the player-to-player trade aspect of the game. This was done to simplify both the implementation of the game logic as well as the learning process of the Q-learning based agent. The removal of this element profoundly shapes the way the Q learning agent learns: without being able to rely on player-to-player trading as a significant and accessible means of resource acquisition, learning agents must prioritize strategies aimed at acquiring as diverse a set of resources as possible (at the expense of other non-resource-focused strategies). The second simplification was to prohibit players from playing their development cards at the beginning of the turn before rolling the dice. This is an important aspect of the gameplay since having the option of playing a knight card before rolling the dice can give the current player the opportunity of freeing a board hex currently blocked by the robber. This part of the gameplay was left out in an attempt to simplify the state machine. It should be noted that compared to the first simplification, this simplification is not as significant in determining the strategy adopted by the learning agents. [3]
Game Framework

4.1 Overview

The modular design of a Python-based game framework was crucial to both of our project objectives. We divided our framework into three major components: the underlying game board, the rules-based game logic and the game player(s).

4.1.1 Game Board

The game board component encapsulates all of the information associated with the board, including the hexagon tiles, tile vertices, tile edges and the sea ports. The game board was implemented as an undirected graph of vertices linked together by edges, making it relatively straightforward to implement the game logic on top. We further added the ability to generate a random board layout subject to the following constraints:

1. No 6 or 8 token can be adjacent to another 6 or 8 token.
2. No number can be adjacent to itself
3. No resource can have more than one of 6 or 8 number tokens.
4. At least one mountain must have a 5, 6, 8, or 9 token.

These constraints were enforced in order to give all players equal chance at winning the game irrespective of the initial turn sequence.

4.1.2 Game Logic

In order to allow efficient state-by-state game simulation by learning agents, we implemented the game logic as a state machine. The most significant states in our machine are as follows:

1. **INIT**: Randomly determine the first player to go.
2. **FIRST_SETTLEMENT/SECOND_SETTLEMENT**: Allow current player to place their first and second settlements as part of the initial snake phase.
3. **FIRST\_ROAD/SECOND\_ROAD**: Allow the player to place his or her first and second roads as part of the initial snake phase.

4. **ROLL\_DICE**: Simulate player’s dice roll.

5. **BUILD\_OR\_DEV**: After the dice has been rolled, the player can choose to build a settlement, upgrade a settlement to a city, buy a development card, or play a previously bought development card.

6. **MOVE\_ROBBER**: Move the robber token to a new board hex.

7. **ROB\_PLAYER**: Steal a resource card from a player who has a settlement or city on the hex on which the robber token has just been been placed.

8. **DISCARD**: When a 7 is rolled, all players with more than 7 resource cards in hand must discard half of their resource cards.

9. **ENDED**: Game has ended.

The following simplified state diagram shows the general control flow.
4.1.3 Player

In order to easily plug in different agents into our framework, we designed a base player class from which all of our players are derived. The inheritance graph for our players is shown in Fig.2.

4.2 Performance Benchmarking

In order to get a better idea of our framework performance, we benchmarked its performance using our least computationally intensive agent, the Random Player. On an average PC, the framework is able to simulate 52 games in 1 second, making the training and testing of learning agents on an ordinary computer feasible.
User Interface

We designed and implemented a pygame-based user interface that allows the user to not only play games against our AI agents but also watch AI agents play against each other. The latter proved to be especially useful during the design and refinement of game heuristics since it provided us with insight into the weaknesses of our agents’ strategies. The user is also provided with the ability to pick their player color at the beginning of the game. At this time, the user may also specify their preferred animation speed (slow, medium or fast) to shorten or prolong the move-making process of other agents in the game.

During the gameplay mode, all of the user’s legal moves are highlighted on their turn in order to allow for seamless interaction with the UI. Furthermore, the UI also periodically provides easy-to-understand instructions during the move-making process to facilitate an
unfamiliar player.

Intelligent Agents

6.1 Existing work

Our work builds upon previous investigations of computational intelligence techniques on the Settlers of Catan. Previous works by Pfeiffer and Szita et al. have investigated the applicability of computational intelligence techniques to the game and yielded promising results. In *Monte-Carlo Tree Search in Settlers of Catan*, Szita et al. applied Monte-Carlo Tree Search in their implementation of an intelligent agent to competitively play Catan. The domain-agnostic nature of MCTS was combined with domain-specific knowledge using virtual wins in the tree search phase of MCTS to give larger weights to promising actions. [5]

Pfeiffer, on the other hand, employed hierarchical reinforcement learning to learn high-level strategies and low-level policies for the game. The complex state-action space was made manageable through the use of feature approximation with model trees. High level strategies were divided into smaller behaviors for which independent policies were learned. In our work, we pursue a similar but distinct approach that implements a Q learning agent
using linear approximation combined with a priori heuristic-based knowledge to develop an agent that can rival a human’s performance.

6.2 Random Agent

We first created a Random Agent that randomly picks a move from the set of all legal moves. While this approach is clearly very naive, the Random Agent served as a basis for comparison during the design and benchmarking of other agents.

6.3 Heuristic Agents

We implemented a total of 3 heuristics-based agent, namely a naive agent, a smart agent, and an expert agent. A distinction was created between these three agents to get a better sense of the effectiveness of using heuristics during various parts of the game.

6.3.1 Selected Heuristics

We implemented a number of heurstics to improve our agents’ performance, some of which are as follows:

1. Move Selection: This heuristic identifies which move type to make. It generally prioritizes building cities and settlements over roads since cities and settlements attain victory points whereas roads do not.

2. Place Settlement or City: This heuristic identifies the optimal vertex for the placement of the initial settlements. It seeks to maximize the diversity of resources (i.e. the vertex that the settlement is placed on should ideally be adjacent to three board hexes with different resource types) as well as the probability that the number token for the corresponding board hexes will be rolled.

3. Place Road: This heuristic identifies the optimal edge for the placement of a road. It seeks to move toward those open vertices for settlement placement that allow the player to extend their longest road.
4. **Play Development Card**: This heuristic identifies the opportune moment to play one of the player’s existing development cards. It keeps all victory point cards hidden until it accumulates enough to win the game. It also prioritizes playing knight cards over other cards to allow the player to seize Largest Army.

5. **Move Robber**: This heuristic identifies the ideal board hex to move the robber token to after a knight is played or a 7 is rolled. It first identifies the strongest player (player with the greatest number of victory points) and then targets their board hex that is most likely to be rolled while taking into account the damage that can be done to the other players.

6. **Discard Resources**: This heuristic identifies the ideal distribution of resource cards to discard. It discards the cards that the player is most likely to get again based on the player’s city and settlement placements as well as the probability of the rolls.

7. **Port Trade**: This heuristic is used to identify the optimal trade move for the current player. Given the player’s resource cards, the heuristic first determines the different meta actions, namely `build-city`, `build-settlement`, `build-road`, `buy-development-card`, that the player can make on this turn after acquiring the necessary cards via trading. If any of these meta actions is possible, the heuristic first selects a meta action (generally prioritizing building cities and settlements over building roads and buying development cards) and then makes the appropriate trade.

### 6.3.2 Naive Heuristic Agent

The naive heuristic agent utilizes heuristics only during the initial snake-phase. This agent was created as a benchmark for the Q Learning agent since the Q learning agent only employs reinforcement learning during the turn-based part of the game, and relies on the same heuristics as the Naive Heuristic Agent during the initial snake-phase. This makes the naive agent perfect for analyzing the performance advantage gained from using learning and heuristics during the turned-base phase.
6.3.3 Smart Heuristic Agent

The smart heuristic agent was built on top of the naive one and employs all of the aforementioned heuristics except for port trade. This heuristic was left out in an attempt to isolate the importance of making strategic port-trade moves.

6.3.4 Expert Heuristic Agent

The expert heuristic agent extends the smart heuristic player by adding the port-trade heuristic. This agent served as the main competitor during the training phase of the Q Learning agent.

6.4 Q Learning

6.4.1 Overview

The use of Reinforcement learning techniques for developing computationally intelligent agents for board games has gained popularity over the last two decades. More recently, the remarkable performance of AlphaGo against human experts has revived the interest in the use of function approximators to approximate the RL value function. [1]

Reinforcement learning techniques are typically aimed at training an agent to learn a value function for every game state by playing a large number of games. Q-learning is a type of reinforcement learning in which the agent learns a value function, Q, for all state, action pairs. At each time step, the Q values are updated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$$ (1)

Q learning is an off-policy learner; the learned Q-value function directly approximates the optimal action-value function, independent of the policy being followed.

6.4.2 Linear Approximation

Since Settlers of Catan has a very large state-action space, it is very difficult for Q learning to converge and learn competitive playing strategies within a reasonable timeframe. We
overcame this issue through the use of a linear function approximator.

Q learning with linear function approximation involves learning a set of weights \( w \), instead of the traditional state-action value function. Corresponding to every state-action pair \((s, a)\) in the typical algorithm, linear approximators use a real-valued feature vector \( x(s, a) = (x_1(s, a), x_2(s, a), ..., x_d(s, a))^T \) with the same number of components as \( w \). Linear methods approximate the state-action value function by the inner product between \( w \) and \( x(s, a) \):

\[
\hat{v}((s, a), w) = w^T x(s, a) = \sum_{i=1}^{d} w_i x_i(s, a)
\]  

(2)

The vector \( x(s, a) \) is called a feature vector representing state \( s \). Each component \( x_i(s) \) of \( x(s, a) \) is the value of a function \( x_i : S \rightarrow \mathbb{R} \). We think of a feature as the entirety of one of these functions, and we call its value for a state \( s \), a feature of \( s \). [4]

At each step of the learning process, the weights are updated as follows:

\[
w_i \leftarrow w_i + \alpha (r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)) f_i(s, a)
\]  

(3)

Since the linear approximator has no local minima, the linear approximator tends to learn more robustly and quicker than traditional Q learning. [6]

### 6.4.3 Agent Design

The success of a linear approximation agent is contingent on the features used to summarize the important characteristics of the game state. It was therefore very important to use features that captured the essential aspects of a game state that guide the strategic move-making process for an expert human player. After discussing the strategic aspects of the game with a large number of experienced players, we were able to narrow down our features to the following

1. Number of knights needed to become largest army player
2. Number of roads needed to become longest road player
3. Victory points of all players

4. Resource production controlled

5. Resource cards owned

6. Ports controlled

7. Development cards owned

8. The ability to build a settlement (ignoring any resource card requirement)

Due to the size of the action space, we designed our agent to learn meta-actions such as \textit{build-road}, \textit{build-settlement}, \textit{play-dev-card} etc. instead of learning the primitive game moves. We then employed the heuristics used by our heuristics-based agents to map the learned meta actions to actual game moves (e.g. \textit{place-settlement-on-vertex-17}).
6.5 Results

(a) Win Performance against Random agents

(b) Win Performance against Naive agents

(c) Win Performance against Smart agents

(d) Win Performance against Expert agents

(e) Win Performance against QLearning agents

Figure 4: Win Performance Comparison

We benchmarked the performance of our players by pitting each agent against 3 players of another type for 5000 games and recorded the winning percentages and the average and
standard deviation of the final victory points. As expected, our results indicate that the expert heuristic player is the strongest of all non-learning players, both in terms of win percentages and average victory points. Our Q learning agent is however able to beat the
performance of all other players, including the expert heuristic agent. When played against three expert players, the learning agent was able to win 38.25% of the time with an average of 7.56 victory points. Since the heuristics used for mapping meta-actions to primitive moves were the same across both players, this showed the learning agent’s superior performance in selecting meta-actions during the turn-based part of the game.

Moreover, the relative performance of the Expert and Smart Heuristic agents demonstrate that making strategic trade moves can lead to an approximate 7.5% increase in winning percentage. This is to be expected since experienced human players, in the absence of inter-player trading, tend to rely heavily on port-trading, especially during the later parts of the game. It is also interesting to note here the remarkable performance gain achieved by implementing heuristics during just the initial snake-phase of the game. Even though the initial phase consists of only four moves, these few moves can easily determine the eventual fate of the game as evidenced by the 73.0% winning percentage of the Naive player when played against three random players.

**Future Work**

We would like to extend the Pysettlers framework using socket programming to give users the ability to play against other users over the network. We would also like to extend the existing user interface functionality to provide users with more information about the current game state (e.g. number of knight cards played by other players) in order to allow them to make better-informed moves. We also look forward to exploring the possibility of incorporating player-to-player trading both in our game framework and our AI agents.

Future work in the agent-design domain may focus on the learning of high-level strategies to complement the learning of low-level meta-actions. We would also like to explore the possibility of using neural networks as non-linear function approximators to better approximate the state-action value function.
Distribution of Work

Since the existence of a functional game framework was a prerequisite for the design and implementation of our AI agents, Chris and I had to first concurrently work on the game framework. During this phase, Chris focused on handling the game logic associated with move-making while I worked on the implementation of the state machine for the game. Once this was completed, we needed the ability to test our framework and so we decided to work together on the UI implementation. During this phase, Chris worked on creating and refining the visual icons while I focused on their placement on the screen.

During the second half of the project, Chris attended to the design and refinement of our game heuristics while I focused more on their incorporation into our framework. After all of the heuristic agents had been implemented, I first designed features for our Q learning agent and subsequently implemented them. At this time, Chris focused on the agent training, tweaking the hyperparameters (learning rate, exploration rate, and discount factor) to achieve optimal performance.
References


http://www.gm.fh-koeln.de/ciopwebpub/Kone15c.d/TR-TDgame_EN.pdf