Computational Intelligence for

Settlers of Catan

Christopher Kim, YC ‘19
Yale University

Omar Ashraf, YC ‘19
Advisor: James R. Glenn, Ph.D.
# Table of Contents

Abstract 2

Introduction and Game Mechanics 3
  Simplifications 4

Part I: Game Implementation 4
  Game Logic 4
  User Interface 8

Part II: Computational Intelligence 9
  Existing Work 9
  Implementation of Agents 10

Results 12

Discussion/Future Work 17

References 17

Distribution of Work 18
**Abstract**

*Settlers of Catan* is a stochastic, multi-player board game in which players assume the roles of settlers, attempting to build and develop holdings while trading and acquiring resources. Previous works by Pfeiffer and Szita et al. have investigated the applicability of computational intelligence techniques to this game and yielded promising results. In this work, we write a functional Python-based *Settlers of Catan* framework and build on the work of Pfeiffer and Szita et al. by applying and testing reinforcement learning strategies to build a competitive AI agent to play this game. The implementation of the game was divided into two main components: a front-end user interface that ensured intuitive, seamless, and correct gameplay and the game framework that instantiated the game logic and was decoupled from the application to run automated simulations to manually play the game and benchmark agent performance. The game logic involved the implementation of tasks such as identifying particular game states that the game would move between, generalizing all possible moves that could be made, creating random board layouts for the game to be played on, and writing validity checks to ensure robustness and correctness of gameplay. The user interface was implemented through Pygame and offered different modes to provide flexibility in debugging and identifying areas of heuristics improvement. Once this stage was complete, computational intelligence techniques were applied in the design and implementation of several artificial intelligent agents and the performance of these agents were evaluated against one another. The main approach that was taken in the design of the principal agent was implementing high-level reinforcement learning that learned meta-actions which were then mapped to primitive game moves using heuristics. The findings of this work suggest that a Q learning approach using linear approximation is a viable means of achieving a strong computational intelligence agent that can compete against human players.
Introduction and Game Mechanics

This work seeks to address two main goals: 1) develop a functional Python-based *Settlers of Catan* (henceforth referenced to as *Catan*) framework using the Pygame library, and 2) apply computational intelligence concepts to design and implement an artificial intelligence agent to competitively play *Catan*.

*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, lumber, wool, 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 through the following ways:

1. **Settlements** - Settlements are placed on the corners of hexagonal tiles and is 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 main 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. Similar to *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 draws a victory point card, he or she is awarded a single victory point.

The game is divided into two phases: an initial snake phase where players place an initial settlement and an initial road in a particular order and a second settlement and second road in the opposite order to ensure fairness. Following this phase is the turn-based gameplay where the start of each player’s turn is marked by the rolling of the dice.
Simplifications

There were two major simplifications made for the sake of easier learning. The first simplification was eliminating the player-to-player trade aspect of the game. The rationale behind its removal was due to the implementation of this functionality posing a significant hurdle in the implementation of the game logic as well as the implementation of the Q learning agent. The removal of this element profoundly shapes the way that the Q learning agent learns: without being able to rely on player-to-player trading as a significant and accessible means of resource acquisition, the computational intelligence agents may be more inclined to pursue a policy in which they seek to acquire as diverse of resources as possible and forsake other 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 a significant aspect of the gameplay because if a player is granted the option of playing a knight card before the dice has been rolled, he or she would take it because it frees a blocked board hex. It should be noted that this simplification is not as significant in determining the strategies of the agents and influencing the strategy adopted by the learning agents as is the first simplification.

Part I: Game Implementation

Game Logic

The implementation of the Catan framework in Python required background research in and familiarity with Pygame, a cross-platform set of Python modules designed for writing video games. An object oriented programming approach was pursued in the development of the framework for the sake of modularity and cleanliness, and the framework was comprised of the following three classes:

- **Game class**
  - The *Game* class includes attributes such as players, board, development cards, resource cards, and game interface
  - Select Classes & Methods
    - The *next_position* method retrieves the current game state and updates the game state according to the flow of the gameplay
    - Select Game States (see Figure 1)
      - **INIT** - randomly determines the first player and assigns turns
      - **FIRST_SETTLEMENT/FIRST_ROAD** - player places first settlement and first road
      - **SECOND_SETTLEMENT/SECOND_ROAD** - player places second settlement and second road which initializes the turn-based gameplay
      - **ROLL_DICE** - simulate player’s dice roll at the beginning of the turn
      - **DISCARD** - if a 7 is rolled and the number of resource cards in a player’s hand is greater than 7, discard half of the player’s resource cards
      - **MOVE_ROBBER** - move the robber token to any board hex
- **ROB PLAYER** - steal a resource card from a player who has a settlement or city on the hex that the robber token has been placed on
- **BUILD_OR_DEV** - player can choose to build a settlement, upgrade a settlement to a city, buy a development card, or play a previously bought development card
- **ENDED** - game has ended (i.e. a player has reached 10 victory points)

---

**Figure 1 - Finite State Machine that tracks all possible game states**

- The *make_move* method takes in a move and the player making the move and updates the *Player* object, *Board* object, and *Game* object accordingly to reflect the move that the player made
- Select Moves
  - **SETTLEMENT** - decrement the number of remaining settlements that can be built, create a new settlement on a given vertex, increment player’s number of victory points, recompute longest road length if the
placement of the new settlement disrupts an existing longest road, recompute player’s resource cards

- **ROAD** - decrement the number of remaining roads that can be built, create a new road on a given edge, compute length of player’s longest road following the placement of the new road, assign longest road to the appropriate player, recompute player’s resource cards
- **CITY** - decrement the number of remaining cities that can be built, upgrade a previously existing settlement on a given vertex to a city, increment player’s number of victory points, recompute player’s resource cards
- **BUYDEV** - pop a development card from the randomly shuffled list of all development cards and append the card to the player’s list of development cards, recompute player’s resource cards
- **PLAYDEV** - if a victory point card is played, increment player’s number of victory points by one; if a knight card is played, enter the MOVE_ROBBER game state, recompute largest army size, and assign largest army to the appropriate player; if a monopoly card is played, monopolize a resource by gathering all existing resource cards in play and adding them to a player’s hand; if a year of plenty card is played, enter YEAROFPLENTY game state; if a road building card is played, enter FIRST_ROAD_MOVE/SECOND_ROAD_MOVE game state
- **PORTTRADE** - process the appropriate port transaction that allows a player to exchange $x$ number of a particular resource with the bank for $y$ number of another resource

The is_valid_move method checks for each move type as defined above whether the given move is valid by subjecting each move to appropriate gameplay constraints

- Select Validity Checks
  - **SETTLEMENT** - vertex to build on is not occupied, no settlements are built on directly neighboring vertices, player has sufficient resources, and player has not built the maximum number of settlements buildable
  - **BUYDEV** - player has sufficient resources and the development card deck is not empty

- **Board** class
  - The Board class includes attributes such as the number of hexes, edges, and vertices associated with a particular Catan board; it instantiates a grid of hexagonal tiles that are randomly assigned appropriate dice rolls and tile types
  - Select Classes & Methods
    - **HexVertex** class
      - A HexVertex (hexagon vertex) is initialized with bordering edges, $x$ and $y$ coordinates, neighboring vertices, adjacent hexes, building, and port edge
      - The can_build_settlement method validates whether a settlement can be built on a particular vertex
- **HexEdge class**
  - A `HexEdge` (hexagon edge) is initialized with start and end vertices, adjacent hexes, building, and port edge
  - The `can_build_road` method validates whether a road can be built on a particular edge

- **BoardHex class**
  - A `BoardHex` (board hex) is initialized with edges, vertices, neighbors, tile type defining the resource that a given hex will produce, and a number token that corresponds to the appropriate dice roll that must be rolled to collect the resource of production

- **Player class**
  - The `Player` class includes attributes such as color, number of victory points, knights played, length of longest road, development cards, resource cards, existing settlements, existing cities, and existing roads
  - Select Classes & Methods
    - The `get_build_settlement_moves` method returns all possible settlements that can be built on existing vertices by the given player assuming resource and board constraints
    - The `get_build_road_moves` method returns all possible roads that can be built on existing edges by the given player assuming resource and board constraints
    - The `get_buy_dev_moves` method returns whether it is possible for a player to buy a development card assuming resource constraints and non-emptiness of the development card deck
    - The `get_play_dev_moves` method returns whether it is possible to play a previously bought development card that exists in a player’s deck
    - The `get_port_trade_moves` method returns whether it is possible to perform an appropriate port or trade transaction given a player’s existing port settlements
    - The `compute_longest_road_length` method computes the length of a player’s longest road by first computing a player’s road segments using breadth-first search and for each road segment, finding the segment’s start position and computing the length of the segment using depth-first search
    - `HumanPlayer` class inherits from the `Player` class and interacts with the user interface to make the desired moves
    - `RandomPlayer` class inherits from the `Player` class; its move is randomly selected from the set of all distinct possible moves

- **Utils**
  - Contains definitions for select classes (see below) as well as enumerations for game state, resource type, tile type, player color, building type, development card type, move type, and port type
  - Select Classes
    - `Dice` class
    - `Road` class
      - The `Road` class includes attributes such as color and edge
- **Building class**
  - The **Building** class includes attributes such as type (i.e. settlement or city), color, and vertex

- **Robber class**

**User Interface**

The interface for the human player to interact with the program was implemented using Pygame. The user interface is comprised of two different modes: 1) a **watch** mode that allows the human player or user to watch and track the progress of games in which the artificial intelligence agents play against one another and 2) a **play** mode allows human players to play directly against AI agents to test their performance. The flexibility of having these two modes was useful for the purposes of debugging and identified areas of weakness that the agents had, in order to improve upon their heuristics. There are many features built into the user interface (see **Figures 2 and 3**). One allows the user and/or player to adjust the speed of the animation for quicker or slower play. This allows the speed of the game play to be adjusted to the comfort and liking of the user. We went to great lengths to ensure that the interface was intuitive to the user: such efforts included highlighting all available moves on the board so that the user could know exactly what moves were available, automatically showing all possible combinations of moves to the user, and animating the dice roll. A text box was also implemented at the top of the game window to guide the users throughout the gameplay by providing instructions during the move-making process and informing players of valid or illegal moves.

![Figure 2 - Screenshot of the User Interface in Watch mode](image)
Part II: Computational Intelligence

Existing Work

Our work builds upon previous investigations of computational intelligence techniques on the *Settlers of Catan* game. Previous works by Pfeiffer[3] and Szita et al.[1] have investigated the applicability of computational intelligence techniques to this 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*. They combine this with domain knowledge using virtual wins in the tree search to give larger weights to actions that looked promising. Pfeiffer employs a hierarchical reinforcement learning with model trees to learn high-level and low-level policies and overcomes the problem of having to learn the Q value for every state-action pair by utilizing feature approximation using model trees and dividing the whole strategy into smaller behaviors in which independent policies were learned. 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.
Implementation of Agents

Part II of this work involved the implementation of artificial intelligence agents that could be trained against other AI players as well as human players to play Catan competitively. There were 5 primary computational intelligence agents that were implemented:

I. Random agent
II. Naive Heuristic agent
III. Smart Heuristic agent
IV. Expert Heuristic agent
V. Q Learning agent

The first and most primitive computational intelligence agent that was implemented was the Random agent. This agent makes random decisions by inheriting from the Player class and randomly selecting a single move from the list of all distinct possible moves.

The second agent that was implemented was the Naive Heuristic agent. The Naive Heuristic agent replicates the behavior of the Random Agent in randomly selecting a move from the list of all possible moves but employs a heuristic for choosing the two initial settlements before the turn-based gameplay is initialized. The reason for implementing this agent was to benchmark more intelligent and sophisticated AI agents against this one to evaluate performance.

The third agent we implemented was the Smart Heuristic agent. This agent was built on the previous Naive Heuristic agent; additional heuristics were implemented and extended jointly between myself and my partner for this agent.

The fourth agent we implemented was the Expert Heuristic agent. The Expert Heuristic agent replicates the behavior of the Smart Heuristic Agent but implements a heuristic that allows for trading with ports and the bank.

Select Heuristics:

- Move Selection
  - This heuristic identifies which move type to make
  - It prioritizes building cities and settlements over roads since cities and settlements obtain for a player victory points whereas roads do not

- Place Settlement / Place City
  - This heuristic identifies the optimal vertex for the placement of the settlements and selection of settlements to upgrade
  - It seeks to maximize the diversity of resources (i.e. the vertex that the settlement is placed on is adjacent to three board hexes of different resource types) as well as the probability that the number token for the corresponding board hexes will be rolled

- Place Road
○ This heuristic identifies the optimal edge for the placement of a road
○ It seeks to move toward open vertices for settlement placement that allow the player to extend his or her longest road

● 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 prioritizes playing knight cards over other cards to seize Largest Army

● Move Robber
○ This heuristic identifies the ideal board hex to move the robber token to when a knight is played or when 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 and how much damage it can do to the other players

● Discard Resources
○ This heuristic identifies the ideal distribution of resource cards to discard
○ It discards the cards that it is most likely to acquire again based on the probability of the dice rolls

The fifth and final agent we implemented was the Q learning agent. Q learning is a type of reinforcement learning in which the agent learns a value function, Q, for all state-action pairs. The learned Q-value function directly approximates the optimal action-value function, independent of the policy being followed. Because Settlers of Catan has a very large state and action space, it makes it difficult for Q learning to converge and learn appropriate playing strategies within a reasonable training period. Our solution to this problem was to use a linear function approximator. The agent will learn a set of weights instead of the traditional Q-value function, and there will be a real-valued feature vector with the same number of components as the weight corresponding to every game state-action pair. The agent learns meta-actions such as build_a_road, build_a_settlement, and buy_a_development_card; it then uses the appropriate heuristics to map these meta-actions to game moves.
Results

Figure 4 - Graph of the Win Percentages of Naive, Smart, Expert, and Q Learning Agents against 3 Random Agents averaged over 5000 trials

Figure 5 - Graph of the Victory Points Achieved by Naive, Smart, Expert, and Q Learning Agents (and their standard deviations) at the end of the game against 3 Random Agents averaged over 5000 trials
Figure 6 - Graph of the Win Percentages of Random, Smart, Expert, and Q Learning Agents against 3 Naive Agents averaged over 5000 trials

Figure 7 - Graph of the Victory Points Achieved by Random, Smart, Expert, and Q Learning Agents (and their standard deviations) at the end of the game against 3 Naive Agents averaged over 5000 trials
Figure 8 - Graph of the Win Percentages of Random, Naive, Expert, and Q Learning Agents against 3 Smart Agents averaged over 5000 trials.

Figure 9 - Graph of the Victory Points Achieved by Random, Naive, Expert, and Q Learning Agents (and their standard deviations) at the end of the game against 3 Smart Agents averaged over 5000 trials.
Figure 10 - Graph of the Win Percentages of Random, Naive, Smart, and Q Learning Agents against 3 Expert Agents averaged over 5000 trials

Figure 11 - Graph of the Victory Points Achieved by Random, Naive, Smart, and Q Learning Agents (and their standard deviations) at the end of the game against 3 Expert Agents averaged over 5000 trials
Figure 12 - Graph of the Win Percentages of Random, Naive, Smart, and Expert Agents against 3 Q Learning Agents averaged over 5000 trials

Figure 13 - Graph of the Victory Points Achieved by Random, Naive, Smart, and Expert Agents (and their standard deviations) at the end of the game against 3 Q Learning Agents averaged over 5000 trials
Discussion/Future Work

We ran simulations to test the performance of each agent (Random, Naive Heuristic, Smart Heuristic, Expert Heuristic, and Q Learning) against three players of another, singular type. We pitted the Naive, Smart, Expert, and Q Learning agents against three Random agents; the Random, Smart, Expert, and Q Learning agents against three Naive agents; the Random, Naive, Expert, and Q Learning agents against three Smart agents; the Random, Naive, Smart, and Q Learning agents against three Expert agents; and the Random, Naive, Smart, and Expert Learning agents against three Q Learning agents. We recorded the win percentages, victory points obtained at the end of each game, and standard deviation for each agent over the span of 5,000 trials and averaged the final numbers. These numbers serve as indicators of performance. The results strongly suggest that the Q Learning agent is the strongest player amongst the various agents in terms of both win percentage and average victory points. It defeats the Expert Heuristic agent approximately 40% of the time and averages 7.56 victory points per game (see Figures 10 and 11). Because we know that the Q Learning agent employs the same heuristics as the Expert Heuristic agent, we can conclude that the learning agent is superior in its selection of meta-actions during the turn-based gameplay aspect of the game.

For the game framework, future work may involve hosting the framework over a network to expand the ability of players to play against other players. We also seek to extend the existing user interface functionality to allow the user to gain more information about the current game state and to incorporate player-to-player trading.

For the computational intelligence, future work will involve learning high-level strategies to complement the learning of low-level meta-actions and adopting neural net based function approximators to better approximate the state-value function using non-linear functions.

References


Distribution of Work

Since the existence of a functional game framework was a prerequisite for the design and implementation of our AI agents, Omar and I had to first concurrently work on the game framework. During this phase, I focused on the handling the game logic associated with move-making while Omar 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, I worked on creating and refining the visual icons while Omar focused on their placement on the screen.

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