Solving Catan

Sam Schoedel, Sam Triest

March 2023

Abstract

We present a concrete, semi-functional implementation of a program that plays (part of) a Catan game.

1 Introduction

Settlers of Catan is a multiplayer board game where players barter for limited resources to dominate the fictional island of Catan. We're going to assume anyone reading this already knows the rules. If not, it really doesn't matter at all. It is a game of nearly perfect information, so most information needed to play the game can be determined from a visual inspection of the board. In the beginning of the game, players take turns placing two settlements each between resource tiles. Initial settlement placement has the largest impact on the game [5]. Because of this, Sam Triest et al. created an algorithm that determines initial settlement placement. To make the algorithm easily usable, Sam Schoedel et al. created a computer vision program that determines the board state from an image of the board. This paper presents the combination of these two projects into one Catanbot^{TM} program.

2 Determining Board State

The basic Catan board is comprised of the elements shown in Fig. 1. It contains 18 resource tiles with numbers, one desert tile, one robber, some ports, and the players' settlements and roads. The board state is determined using a gauntlet of thresholding, hough transforming, classification, and digit recognition.

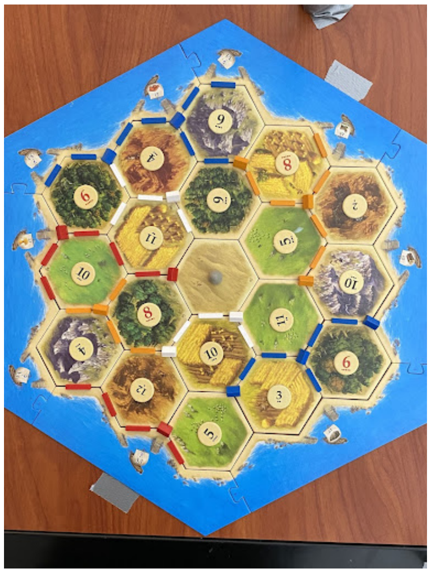

Figure 1: example Catan board.

2.1 Roads and Settlements

A graph of the board was created by finding each number tile using a circle Hough transform and then branching between each circle in a hexagonal pattern. This allowed us to compute each of the green dots shown in Fig. 2. We then just masked out all parts of the image that weren't near expected settlement and road locations filtered based on player color.

Figure 2 shows correctly classified roads and settlements for the orange, blue, red, and white players. The green dots represent potential road and settlement locations. The initial settlement placement logic uses existing settlement and road placement information to determine the next best place to build a settlement.

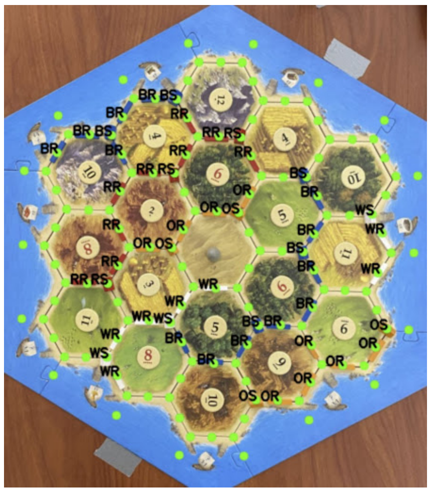

Figure 2: road and settlement detection.

2.2 Resource Tiles

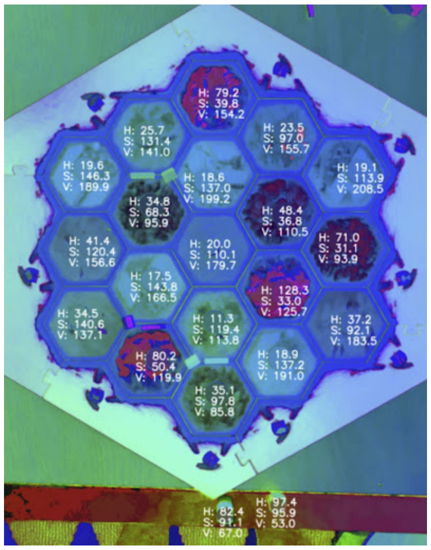
H: 79.2 S: 39.8 V: 154.2
H: 25.7 S: 131.4 V: 141.0
H: 23.5 S: 97.0 V: 155.7
H: 19.6 S: 146.3 V: 189.9
H: 18.6 S: 137.0 V: 199.2
H: 19.1 S: 113.9 V: 208.5
H: 34.8 S: 68.3 V: 95.9
H: 48.4 S: 36.8 V: 110.5
H: 41.4 S: 120.4 V: 156.6
H: 20.0 S: 110.1 V: 179.7
H: 71.0 S: 31.1 V: 93.9
H: 17.5 S: 143.8 V: 166.5
H: 128.3 S: 33.0 V: 125.7
H: 34.5 S: 140.6 V: 137.1
H: 11.3 S: 119.4 V: 113.8
H: 37.2 S: 92.1 V: 183.5
H: 80.2 S: 50.4 V: 119.9
H: 18.9 S: 137.2 V: 191.0
H: 35.1 S: 97.8 V: 85.8
H: 82.4 S: 91.1 V: 67.0
H: 97.4 S: 95.9 V: 53.0

Figure 3: resource detection.

Resources are detected by just hard coding HSV thresholds for each type. Figure 3 shows the average HSV values for each tile with the nominal lighting in my room at the time. The average resource hues and saturations are far enough apart that it works under these conditions. If the lighting changes it completely breaks. Oh well.

2.3 Ports

Ports were hard to classify because the only features they have are the little icons on each ship. Since the photos we used were taken by a phone camera, we solved this problem by color thresholding each ship in the water area, finding the most obvious contours (the ships), and zooming in on them. This gave us a higher-than-necessary image resolution for each icon. After taking a bunch of photos of ports and hand annotating them we were able to train a classifier with about 95% validation accuracy.

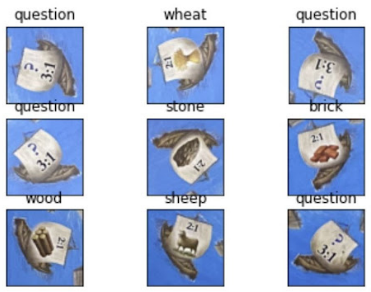

Figure 4: port classification.

Here's some port results. I guess we could have just lied about the accuracy and hard coded these labels, but you'll have to take my word for it. We did hard code the location of each port. hehe

2.4 Numbers

Off the shelf digit classifiers couldn't detect the tile numbers at all, so we opted to spend too many hours hand labeling another dataset and training another classifier with around 95% validation accuracy.

We found each number using a circle Hough transform and cropped the original high resolution image to obtain just the number section, then did some more thresholding using the little dots under each number tile to orient the image vertically. Classification was then done on the vertically oriented, highly zoomed in image. Figure 5 is an example of a board state with correctly classified and annotated numbers.

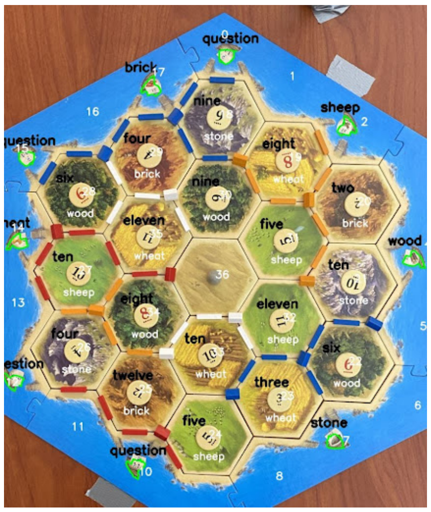

Figure 5: number classification.

2.5 Robber

The Robber's wherever there's no number tile.

2.6 All Together Now

Putting it all together, we get something that looks like Fig. 7, a graphical representation of the same board we took a photo of.

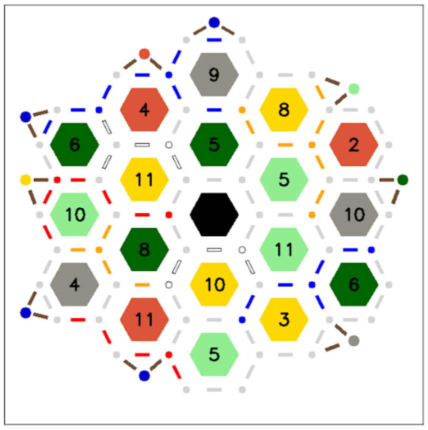

Figure 6: board state recreated.

3 Initial Placement

Disclaimer: I'm writing on behalf of my partner for this part, but he gave me some material he wrote to pull from so we should be all good.

Disclaimer: I wrote the code and I barely know what's going on, so we should be all good.

Let's show some results first:

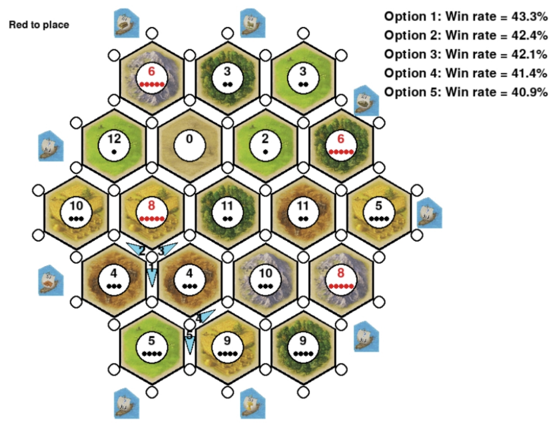
Red to place
Option 1: Win rate = 43.3%
Option 2: Win rate = 42.4%
Option 3: Win rate = 42.1%
Option 4: Win rate = 41.4%
Option 5: Win rate = 40.9%

Figure 7: winrates with red to place.

Our system uses a combination of Monte-Carlo tree search and deep reinforcement learning to choose good initial settlement locations for the game Catan. Shown in Fig. 7 are initial placement suggestions for a particular board.

Ideas from AI have been applied to Catan in the past. One of the earliest instances is Jsettlers [2], an implementation of Monte-Carlo Tree Search for Catan. [3] and [1] both employ reinforcement learning techniques to the main game of Catan. All previous works in this area focus on playing the game of Catan after the initial placements phase. [2] use Monte-Carlo tree search to select actions and thus do not require learned functions for value estimation. [3] and [1] both train function approximators on self-play in order to select optimal actions. [3] use hand-crafted features, while [1] transform the hexagonal board into an image-like grid in order to use convolutional neural networks for value estimation.

3.1 Catan Simulator

We first wrote a GPU-accelerated, vectorized Catan simulator to play Catan games. Using this Catan simulator, we can play like 50 games per second on a NVIDIA 3080.

3.2 Time for an Algorithm

Here's an algorithm for all of your troubles (you may have to zoom in to get the full effect):

```
Algorithm 3: DQN with Alphago-style Self-Play
Input: Untrained Q-function parameters $\theta$, Replay Buffer $\mathcal{D}$, win threshold $\alpha$, discount factor $\gamma$,
       Learning rate $\eta$, Polyak term $\beta$
Output: Trained Q-function parameters $\theta$
$\theta' \leftarrow \theta$
$\theta^* \leftarrow \theta$
while not done do
    $\tau = \text{rollout}(\theta^*)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \tau$
    for train iterations do
        $(s, a, r, s', t, p) = \text{sample}(\mathcal{D})$
        $J(\theta) = \begin{cases} -(Q_\theta(s,a,i) - (r + \gamma \max_{a'} Q_{\theta'}(s',a',i)))^2 & \text{if } i = p \\ -(V_\theta(s,i) - (r + \gamma V_{\theta'}(s',i)))^2 & \text{otherwise} \end{cases}, \forall i \in [0,3]$
        $\theta \leftarrow \theta + \eta \frac{\partial J}{\partial \theta}$
    end
    $z \leftarrow \text{compare}(\theta, \theta^*)$
    if $z > \alpha$ then
        $\theta^* \leftarrow \theta$
    end
    $\theta' \leftarrow \beta\theta' + (1-\beta)\theta$
end
return $\theta^*$
```

Figure 8: A troubling algorithm.

3.3 Self Play and Q-Function Learning

We train a graph neural network to predict the Q functions for each player in a given state. This is accomplished via a method similar to AlphaGo [4] in which the agent plays against earlier versions of its Q function. The Q function is trained to predict the probability of each player winning from a terminal state, where the probabilities are obtained from Monte-Carlo rollouts of the Catan simulator. Fro non-terminal states, we minimize squared Bellman loss.

3.4 MCTS

We then use Monte-Carlo tree search (MCTS) to search for optimal initial placements. When the tree reaches a terminal node (after 8 placements), the value of a node for each player can be computed by rolling out several Catan games from that initial placement. After expanding several thousand nodes, we get a reasonable guess for the settlement placement.

Overall we find that the combination of MCTS and Q-learning outperforms random placements, as well as a heuristically designed placement policy.

Method	Win Rate Against MCTS + QF	Win Rate of MCTS + QF (ours)
MCTS	47.5 %	52.5 %
QF	41.2 %	58.8 %
Heuristic	34.4 %	65.6 %
Random	8.8 %	91.2 %

Figure 9: comparison against other methods.

4 Results

In order to evaluate the efficacy of our method, we seek to answer the following questions:

1. *Can* you build a bot to play Catan?
2. *Should* you build a bot to play Catan?
3. *Did* we build a bot to play Catan?

4.1 Can you build a bot to play Catan?

You sure can.

4.2 Should you build a bot to play Catan?

Depends on who you ask. I beat my friends with the bot once and they said they won't play Catan with me again. Our advisors asked why we wasted lab compute on this. So overall probably yes.

4.3 Did you build a bot to play Catan?

We sure did.

References

[1] Quentin Gendre and Tomoyuki Kaneko. *Playing catan with cross-dimensional neural network*. (accessed: 10.07.2022).

[2] Guillaume Chaslot Istvan Szita and Pieter Spronck. *Monte-carlo tree search in settlers of catan.* (accessed: 10.07.2022).

[3] Michael Pfeiffer. *Reinforcement learning of strategies for settlers of catan.* (accessed: 10.07.2022).

[4] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[5] Me. I said this.