Catan: The Settlers of Python

15-112 Term Project Proposal by David Hwang (dchwang)

Description:

Name: Catan: The Settlers of Python
This project will be a python implementation of the popular board game Catan.

Competitive Analysis:

The Settlers of Python seeks to create a fun Python implementation of Catan. The game will allow at least one player to play a game of Catan. As a result, the game also implements an AI to fill the other player positions. The goal of the game will remain unchanged from the original rules of Catan; in order to win, a player must reach 10 victory points before any other player.

Some competitors have been identified that are similar to this project: colonist.io and notcatan. Colonist.io is a web-based implementation of Catan. It allows the user to play with multiple other players or with an AI. On the other hand, notcatan is a Python implementation of Catan for multiplayer use. It uses Python sockets to allow clients to connect to a host and play from different machines. Among these two projects, colonist.io has more focus on AI than notcatan, while notcatan focuses more on the use of sockets to allow different people to play at once. From these, my project will be similar to the platform of notcatan with an implementation like colonist.io. The main difference between my project and these competitors is that my project will be used on one machine only, rather than multiple machines for multiplayer.

Structural Plan:

Elements (GUI Features):

- **Element** class (/resources/gui/element.py)
  - **Button** class (/resources/gui/button.py)
    - self.draw(): method to draw the **Button** on the screen
  - **Scorecard** class (/resources/gui/scorecard.py)
    - self.draw(): method to draw the **Scorecard** on the screen

Catan Features:

- **Board** class (/resources/game/catan.py)
  - Board Generation (__init__)
    - self.hexBoard: stores the board as a List. Calls generateAxialList().
    - self.hexCount: number of hexes in the entire board
    - self.edges: List to store all **Edge** objects
    - self.generateEdges(): method to create all N instances of **Edge** objects in board. (n = self.hexCount)
    - self.assignEdges(): method to assign the **Edge** instances to indices in self.hexBoard
    - self.assignTypes(): method to randomly assign the **Tile** types
    - self.assignNumbers(): method to assign number tokens fairly, i.e. according to standard Catan rules.
- **Player** class
  - __init__: initialize Player color and point counters, also keeps track of roads
  - self.countRoads(): method to count longest continuous road with self.color
- **Tile** class
  - __init__: initialize Tile type and List of adjacent **Edge**s
  - __repr__(self): prints Tile in a friendly format
- **Edge** class
  - __init__: id: ID value of the **Edge**. Each **Edge** has a unique ID
    - self.road
      - None, if no road on this **Edge**
      - String, indicates the color of the road on this **Edge**
  - __repr__(self): prints **Edge** in a friendly format
  - __eq__(self): enables equality checking of **Edges** based on ID
  - __hash__(self): hash function for **Edge**
- **Node** class
  - __init__: Initializes the Node with no settlement. Mutable to indicate presence of settlement/city. Defines port adjacencies.
  - __repr__: prints **Node** in a friendly format

Config & Settings (constants):

- **Colors** class: contains Pygame Color instances that will not be changed as the app runs
- **Config** class: contains constants that will be used to store settings (i.e. screen size)

Game Instance:

- **PygameGame** class (/pygameFramework.py): modified version of Pygame Framework from mini-lecture
    - **CatanGame** class (/game.py)
        - self.init(): analogous to appStarted()
        - self.keyPressed()
        - self.mousePressed()
        - self.redrawAll()
            - self.drawMenu()
            - self.drawBoard(): Board, Players, Trade, Buy, Resource
            - self.drawPause()
            - self.drawEnd()

Algorithmic Plan:

Catan AI using a Monte Carlo Search Tree algorithm

Starting the Turn:

- Dice is automatically rolled
- Resources received by all players

Selection:

- **Board** is copied to temporary variable
- Legal moves from current state is listed
- Randomly sample (doMove) a subset of the legal moves

Expansion & Simulation:

- Expand the tree by adding a childState to sampled state
- Check if childState is victoryState
- If not, simulate next possible moves from childState

Backpropagation:

- Store move weights (preset value based on likelihood to earn a victory point) of simulated states

Ending the Search:

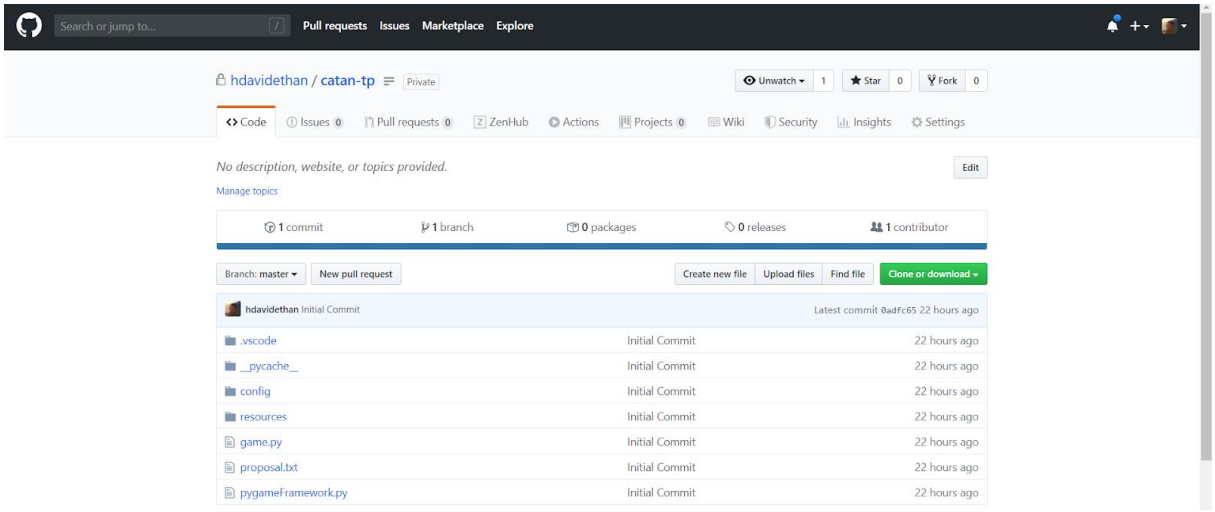
- Check all move weights and find largest value.
- Do that move, store the new state of board and end turn

Timeline:

- Complete Board Controllers, i.e. Nodes and Roads (By 11/20/2019, Wednesday)
- Complete Preliminary Board Representation (By 11/21/2019, Thursday)
- Complete In-Game GUI (By 11/21/2019, Thursday)
    - Board
    - Cards
    - Points
- Complete VP Counting (By 11/22/2019, Friday)
    - Settlements/Cities
    - Largest Army
    - Longest Road
- Complete Preliminary AI Algorithm (By 11/25/2019, Monday)

Version Control Plan:

This project is on Git and is fully uploaded to a private repository on GitHub.



Module List:

1. pygame

TP2 Update:

Since TP1, I have implemented most of the indicated features in the Structural Plan. However, I have slightly changed some of the implementations from the original plan. First, I created a Dice subclass of Element since it was much easier to render them as Elements rather than create another class. In addition, I also added the onClick() method (which does nothing) to the main Element class. Only the Button class modifies this to handle click events on Buttons. I also created a Text class to store fonts and moved many of my utility (Utils class) and math (CatanMath) functions to utils.py. With regard to my AI functionality, I found it much simpler to create a

subclass of Player called AIPlayer. AIPlayer will contain all of the methods to calculate the next move for the AI.