Settlers of Catan AI

By: Jonathan West, Vincent DiPerna, Chiraag Potharaju

Problem Statement and Analysis

- Catan is a popular board game, and has a large user base
- There is a continual push to improve play
- Effective AI Game Playing Agents can help players improve
- This is shown with existing AI agents for other games

Catan Overview

- 2-4 Person Game
- Win by obtaining most “Victory Points”
- Build & expand settlements, harvest resources, play development cards, and trade with or rob other players
- Multiple Hexes (Resources),
  - Number on each hex matters
  - Positioning of hexes matters
- Multiple Actions per Turn (Order matters)

Use Cases

1 Catan Players

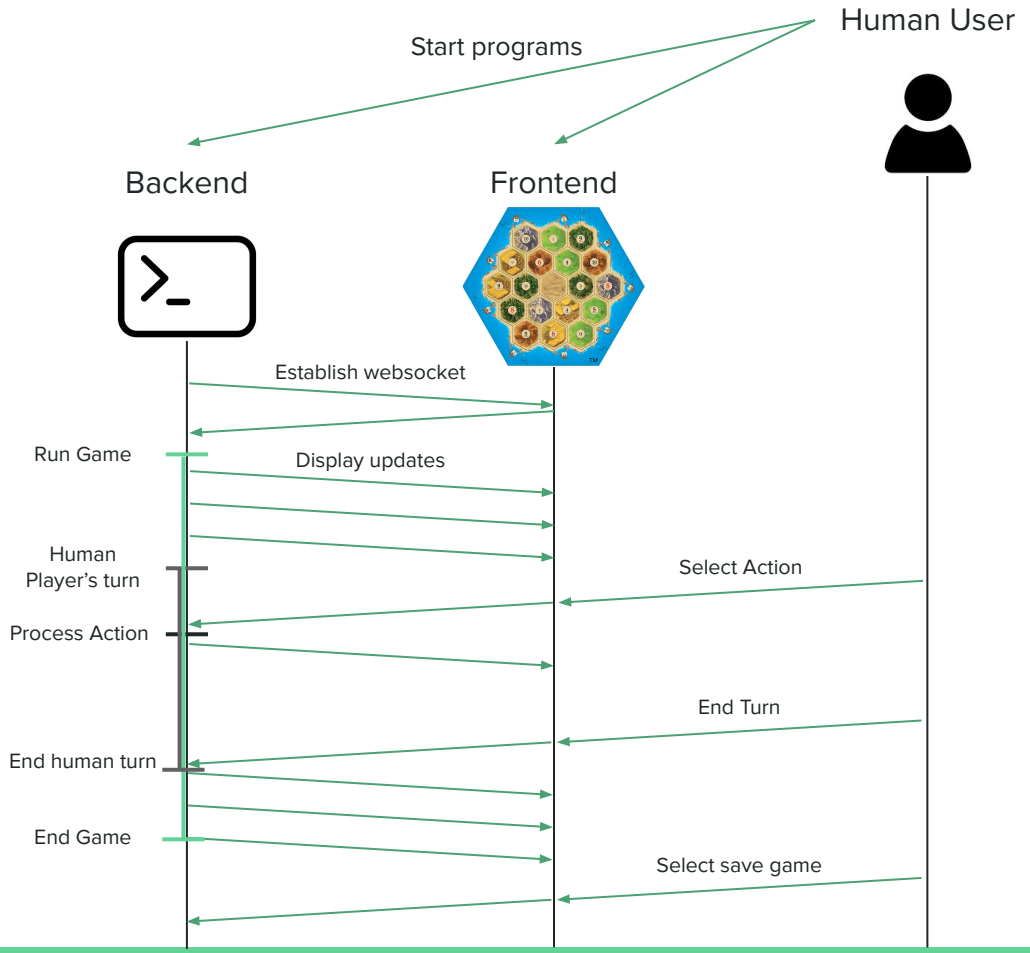
- Test new strategies
- Generate database of games to review
- Practice without needing other people
- Learn new strategies from AI

2 AI Researchers & Developers

- Generate large database of strong Catan games
- Apply similar AI models & techniques to new yet similar applications
- Learn from flaws in design when developing new projects

Basic Design

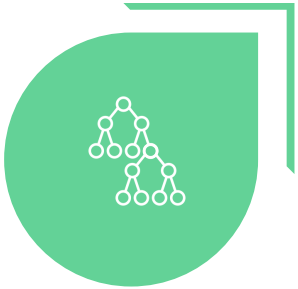
- Python backend game logic
  - Run in terminal with params for settings
- HTML Canvas frontend
  - Logic written with vanilla JavaScript
  - Communication via Websockets
  - Separated frontend logic to keep efficient backend
- AI Agents placed in additional files



AI Algorithm and Model: Considerations

Expectiminimax

- Allows for use with random chance & incomplete info, as opposed to minimax
- Primary AI agent



Monte Carlo Tree Search

- Handles uncertainty well
- More memory efficient than minimax
- Did not have time to implement

Deep Learning

- Could use a neural net as a state evaluation function
- Requires significant time & computational resources

Reinforcement Learning

- Catan AI learns from playing many instances of Catan under different circumstances

AI Algorithm: Expectiminimax

- Like with pacman ghosts, doesn't immediately alternate between two players
- Maximizes/minimizes with same player until they choose to end turn
- After a player ends turn, considers probabilistic dice roll before going to next player's turn in game tree
- Considered several game state features for evaluation heuristic, including player's VPs, opponents visible VPs, number resource & development cards held by each opponent
    - Balance of efficiency with depth of analysis

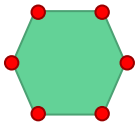
Design Considerations

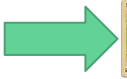
- Balance between efficiency & human convenience
  - Could have used C rather than Python for algorithmic agent
  - Could have made somewhat more compact board state representation
  - Used hybrid bitmaps for board representation
- Flexibility for non-standard game experimentation
  - E.g. don't just assume there are always the same set of resource tiles in the board; allow for this variable to be easily changed
  - Balanced with pragmatic restrictions: e.g. didn't design program to easily allow multiple robbers
- Balancing memory spatial efficiency with computational efficiency
  - Example: Which players get which resources after a given dice roll?

Dice Sum

Hex Indices

Vertices (Building locations)

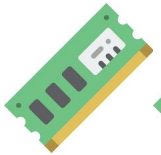


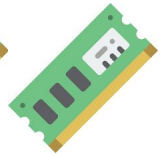


Players get resources

VS.

Direct conversion map in state object, More memory usage

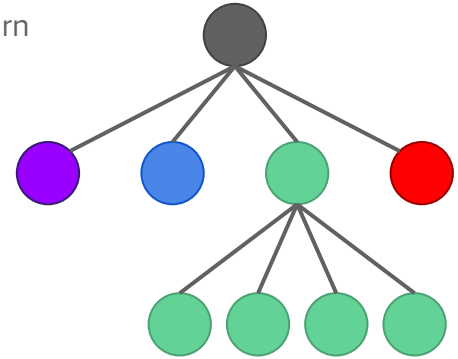




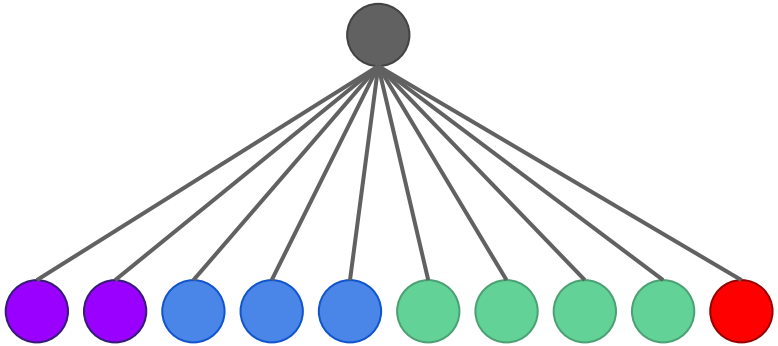
Design Considerations

- Multi-stage decision making can reduce branching factor
  - Allows more specific heuristic implementations
- Generally leaned towards single-stage decision making
  - Used multiple decision stages for a player's turn, but not within a given building action
  - Simpler implementation

Build city
Build Settlement
Build Road
End Turn



Multiple Stages: Considered 8 actions



Single Stage: Considered 10 actions

Results

- Visual display similar to real board
- Allows natural human input using mouse
- Can generate large numbers of saved games for training, analysis, or other purposes
- Can save or load game states to and from .json files, and continue play simply
- Expectimax decent agent versus humans; still being improved upon
  - 100% win rate against random in our testing
- Relatively efficient game state in terms of common operation speed and size (~4000KB), while maintaining readability
- Several simplifying constraints for ease of development

```
# Road map, 1 bit for road locations
RMAP = [
    0,
    0b00000000000000001010101010000000,
    0b00000000000000001000100010000000,
    0b00000000000000101010101010000000,
    0b00000000000001000100010001000000,
    0b00000000000010101010101010100000,
    0b00000000001000100010001000100000,
    0b00000000001010101010101010100000,
    0b00000000010001000100010001000000,
    0b00000000101010101010101010000000,
    0b00000000000000001000100010000000,
    0b00000000000000001010101010000000,
    0]
```

Demonstration

Lessons Learned

- Furthered knowledge on heuristic design with relatively complex game
- Team organization & time management
- Would have implemented at least basic versions of a few more AI agents, but we lacked enough time
- Should have made more specific plan from the outset
- Optimize for common case
- Need to be careful with game simplifications: they can compound and lead to issues
- Should have spent more time on primary evaluation function; the current version was quite simplified for efficiency

Future Work

- Implement more AI agents, notably MCTS
- Continue accounting for parts of the game that were left out
- Improve heuristic evaluation function

Q&A

Try out the program:
https://github.com/jrgranadoswest/ocatan