An AI agent for Settlers of Catan

Sierra Kaplan-Nelson, Sherman Leung, and Nick Troccoli

**Code for gameplay is visible at this Github Repo*

**Please also note, we changed our original project proposal to this revised project proposal (as approved by Jiawei and Ice)*

Introduction

Games are considered important benchmark opportunities for artificial intelligence research. Modern strategic board games can typically be played by three or more people, which makes them suitable test beds for investigating multi-player strategic decision making. In this project, we are using our practice with AIs for adversarial games to implement a game-playing agent for a simplified version of the board game *Settlers of Catan*.

Background

As some background, *Settlers of Catan* is, at its core, a resource-gathering and building board game. Players have a variety of settlements on the board that harvest resources every turn, and those resources can be used to purchase more settlements or other game objects, which ultimately give the player victory points needed to win the game. A picture of a basic game board is shown at right - each hexagonal tile represents a resource, and every tile is also assigned a number between 1 and 12. On a player's turn, the game dice are rolled. Any player with a settlement bordering a hexagonal tile with that number on it receives one resource of that type (cities provide a player with 2 resources of that type). One exception is that if the robber pawn is located on top of a hexagon, that hexagon does not produce resources. The robber must be moved to a new tile by a player whenever that player rolls a 7.

Each player keeps all their resources in their hand. After distributing resources as determined by the dice roll, the current player can then take any number of actions - they may buy a new settlement, city, or road, trade with another player, or buy an action card.

For building, a new settlement must be built on a vertex on the board, it must be at least 2 edges away from any other settlement, and it must be connected to one of the player's other settlements by roads. A new road must be built on an edge on the board, and it must be connected to another one of the player's settlements or roads. A

city (an upgraded settlement, which gets 2x the resources per turn that a settlement does) may be built in place of an existing settlement.

The player may also choose to trade resources with any of the other players. Note that only the player whose turn it is may initiate trades. Finally, the player may choose to buy action cards, which can be played on future turns. Action cards are drawn randomly from the top of the deck. Actions enabled by action cards include stealing other people's resources, gaining extra victory points, and more.

Each new settlement gives the player 3 victory points, and each city gives an additional victory point on top of the 3 for the now-replaced settlement. The first player to get to a certain number of points (10 for 3-player games) wins.

Also, at any point during the game, the person with the longest continuous road (of length >= 6) gets a card worth 2 victory points, and the player with the most knights on the table (one of the possible cards from the deck) gets a card worth 2 victory points. (Ownership of these cards changes over the course of the game).

Input/Output

Our algorithm will be designed to play against either a real human player (where a human inputs the actions he/she would take) or a human-player-heuristic that imitates a real opponent. The success of our algorithm is determined by the win rate and the number of victory points obtained during a game. Therefore, even if our AI does not win, we can still determine its relative success.

Approach

For this project milestone (which we will build on, as outlined in the "Future Additions" section below), we simplified the rules of the game to make it easier to implement, without losing the core strategy of the game. For our simplified game, 3 players play on a 5x5 grid of tiles (a square board, to simplify the complexity of having hexagonal tiles). In this representation, each tile produces a resource and every tile can be either a road or a settlement. On each turn, rather than rolling the dice, each player obtains one resource for every settlement they own. Then, a player can take only one action: either buy 1 settlement, or buy 1 road, subject to the previously mentioned constraints. We also simplify the buying of a settlement to require 4 of any resource, and a road to require 3 of any resource. A player gets 3 victory points for each settlement (not counting their initial settlement), and 1 for each road, and needs 10 victory points to win.

Model

Taking a page from our Pacman assignment, we are modeling the game state using two types of information - a board state, and a player state (for each player in the game). The board state uses a n x n grid to model the board of tiles, and also has a list of settlements and a list of roads. With this representation of the game board, we can execute board queries (adjacent edges to vertices, etc.) very quickly by doing index arithmetic within the grid. Finally, a tile stores the type of resource it produces, as well as what is built on it (if anything) and who controls it (if anyone). With this implementation, we're able to easily keep track of roads, settlements, and tile types.

Each player also has a corresponding player state object that stores state information about that player. This includes 3 lists - a list of the settlements (vertices) owned by the player, a list of roads (edges) owned by the player, and a list of resources owned by the player (representing his/her hand).

Algorithm

In our current implementation, we have implemented a minimax agent that follows the minimax algorithm to determine what actions our AI should take against the opponent. To do this, we are writing an evaluation function to use to determine what actions are optimal. Our evaluation function attributes a utility of +∞ for winning (victory points > 9) and -∞ for losing (another player's victory points > 9 and yours < 10). It also attributes a utility of 3 for every settlement owned by the player, and a utility of 1 for every road owned by the player. Therefore the "utility" for the resources held by a player is absorbed into the utility for building a settlement or road on a future turn. To choose an action on a given turn, our algorithm examines the minimax tree and evaluates the possible actions it could take (currently stopping with a depth of 2).

The action that returns the highest utility allows the minimax agent to move forwards and the board and the player states are consequently updated based on the action taken.

Example + Analysis

We can consider a sample output of our game loop to see how our model and algorithm work (see the board printouts to the right actually generated by our algorithm). As input, the starting settlement positions for each player were hardcoded to simplify the modeling of our algorithm. Each following turn of the game, updates each player agent with the action chosen (e.g. SETTLE or ROAD).

In this example, we pit three player agents against each other. We define our minimax player as agent #0 and random players as agents #1 and #2. It should be noted that each S_i represents a settlement for the i-th player (where i is in range(total # of agents). Similarly, each R_i represents a road owned by a player.

As sample output of our implementation, we note how agent #0 is able to expand and capture settlements. Given a depth of 2, the minimax agent is able to look ahead and realize the investment of resources in settlements leads to a higher utility in the end state.

As agent #1 and agent #2 advance in the game, the number of actions that corresponding to building a road increase with more entry points to building roads. This leads

Minimax (agent = 0)

```
     0   1   2   3   4   5
0 [ --  --  --  --  --  -- ]
1 [ --  --  --  --  --  -- ]
2 [ --  --  S0  --  S2  -- ]
3 [ --  --  --  --  --  -- ]
4 [ --  --  --  --  S1  -- ]
5 [ --  --  --  --  --  -- ]

     0   1   2   3   4   5
0 [ --  --  --  --  S2  R2 ]
1 [ --  --  --  --  --  -- ]
2 [ --  R0  S0  --  --  -- ]
3 [ --  --  --  --  --  -- ]
4 [ --  --  --  R1  S1  -- ]
5 [ --  --  --  --  --  -- ]

     0   1   2   3   4   5
0 [ --  --  --  --  S2  R2 ]
1 [ --  --  --  --  --  R2 ]
2 [ S0  R0  S0  --  --  -- ]
3 [ --  --  --  --  --  -- ]
4 [ --  --  S1  R1  S1  -- ]
5 [ --  --  --  --  --  -- ]

     0   1   2   3   4   5
0 [ --  --  --  R2  S2  R2 ]
1 [ --  --  R0  --  --  R2 ]
2 [ S0  R0  S0  --  --  -- ]
3 [ --  --  --  --  --  -- ]
4 [ --  --  S1  R1  S1  -- ]
5 [ --  --  R1  --  --  -- ]

end state:
     0   1   2   3   4   5
0 [ --  --  S0  R2  S2  R2 ]
1 [ --  --  R0  --  --  R2 ]
2 [ S0  R0  S0  --  --  R2 ]
3 [ --  --  --  --  --  -- ]
4 [ --  --  S1  R1  S1  R1 ]
5 [ --  --  R1  --  --  -- ]
```

agent #1 and agent #2 both towards road expansion, which matches the expected behavior.

Our current implementation models the game with a simple board model (see simpleBoard.py in the appendix) and a single resource type that allows for the acquisition of settlements and roads. *To note*, the minimax algorithm runs into several unresolved bugs that prevents the game loop from completing two cycles without raising an exception. Thus, the sample output given above has been extrapolated given the first two cycles that we can run to match a minimax algorithm's behavior. Despite the unresolved bugs in our implementation, we expect that the successful implementation of our minimax agent would result in the sample output above.

Results and Discussion

Given the successful implementation of our random agent as our baseline algorithm, our random agent played 10 times with a human oracle. The baseline metrics perform as expected - winning 33% of the time against other random agents which is as expected.

Though the minimax algorithm is implemented, the algorithm is blocked by several bugs that prevent that algorithm from being tested successfully against the baseline and oracle algorithms. We expect the minimax algorithm to outperform the random agent 80% of the time, and the minimax to be just under 50% when playing against an experienced human.

Baseline and oracle results for the random and minimax agents

(n=3 agents) *estimated %*	**Random vs. Random** (n=100)	**Minimax vs. Random**	**Random vs. Human** (n=10)	**Minimax vs. Human**
Win Rate	33.3%	*~80%*	0%	*~45%*

Future Extensions

Given that our current model and algorithm focuses mainly on the acquisition of settlements for victory points, we intend to implement several extensions to the model that will allow the program to more accurately reflect the real aspects of the game. The first extension will factor in the dice rolls that determine resource allocation in the game. This requires the implementation of the expectiminimax algorithm. In other words, the expected number and the type of resources that our player obtained will be factored into the algorithm's calculation of future states.

The second extension will allow the action of upgrading settlements to cities. This additional detail will add an additional action and change our evaluation function to include an additional utility of 5 per city owned by a player.

The third extension we would like to incorporate allows for the acquisition the longest-road card, a card that gives 5 victory points to the owner of the longest road on the board. This adds an additional complexity that needs to incorporate players to

lose and gain this card. In more detail, this extension might incentivize agents to retain the longest road by expanding roads into more open places on the board.

The fourth extension is the robber, which would exist in our board state object, and which would prevent resource production on the tile it's on. Incorporating the robber would require another change to our evaluation function, which would be the sum of a negative utility if the robber is on a tile your settlements border and a positive utility if the robber is on a tile your opponents' settlements border. The magnitude of each of these utilities would be directly proportional to the probability that that tile is "rolled" multiplied by the number of your (or your opponents) settlements that border it. These utilities would represent the "cost" or "benefit" to you of the robber's location (e.g. it's really bad if the robber is on one of your settlement's tiles that's really likely to be rolled, but not that bad if the tile is not that likely to be rolled).

Time-permitting, we would like to include trading with other players and purchasing cards from the deck, though these are more ambitious additions that we may not have time to complete. We also hope to create multiple evaluation functions representing different game strategies - for example, prioritizing monopolizing one resource, or prioritizing trying to obtain every resource, or prioritizing building cities vs. building lots of settlements, etc. Additionally, we would like to model a hexagonal board, although the complexity of that is much higher than a basic grid board.

APPENDIX

A description of the different files and classes in our current implementation is described below:

game.py: this file implements the main game loop and defines classes necessary for the game to run - it is also the same file that we run to obtain results

- **Game:** The Game class defines the main game loop within the run method, and also stores the gameState as a data property.
- **GameState:** The GameState class stores all data associated with the board and the player states. Specific to the minimax algorithm, the class generates successor states given a playerIndex and an action. We pass in state objects to our evaluation function to determine the utility given a particular state. This class also implements the getLegalActions() method that will return legal actions given an agentIndex after factoring in the state of the board and other players.
- **Agent:** The Agent class implements the recursive minimax algorithm that returns the best action after optimizing the value of that action given a particular depth and state. It also stores information about the agent's state and contains information that describes the resources, properties, and victory points that an agent holds.

basicBoard.py: this file implements the logic relevant to the board constraints. As described in the model section, our approach to modeling the board represents each road and settlement as a tile element. As in the game, only resources can be obtained from the tiles that have settlements on them.

- **Tile:** The Game class defines properties of a board tile and methods that allow the tile to be captured.
- **BasicBoard:** This is the main class that defines the board logic. It includes methods that allow actions to be applied the board, thus updating the state of the board. The majority of the complexity in this class allows for the retrieval of occupied and neighboring tiles to generate possible actions.

util.py: this file contains class definitions that we will eventually use in our extension.

- **Card:** The card class defines development and victory cards that allow more options and possibilities for player agents to accumulate victory points.
- **Deck:** The deck utilizes python's randomShuffle method to randomize the development cards.
- **Resource:** This is an enum that defines the resources of the game
- **Actions:** This enum defines the different actions an agent can take at a given state in the game.

board.py: This file defines a more complex implementation of the board that allows for hexagonal tiles. It defines vertices and edges that can be captured as settlements and roads, respectively. The logic in this file is derived from the implementation details defined here.