Monopoly Playing Program

1 Introduction

This project aims to build an agent which plays the classic board game monopoly with one or more players. An adjudicator/simulator program will allow us humans to view the stages of the game play and serve as a platform for the agent to communicate with the game.

This progress document aims to report the progress done on this project so far, halfway into the timeline - on both the adjudicator we built and the agent we have made to playoff this game.

2 Adjudicator

The first part of building an agent, is understanding how the agent will be judged. An adjudicator will take in two different agents, and it will simulate a game play for them. This section describes the details about the adjudicator we have built to serve this purpose.

2.1 Design and Architecture

We expect the adjudicator to access two python files, each implementing a common interface as defined by the adjudicator. The adjudicator asks the agents their decisions on each step, and based on that it moves the game forward.

Here is the flowchart of the game that was used to decide the flow of the adjudicator. (Refer Fig.1)

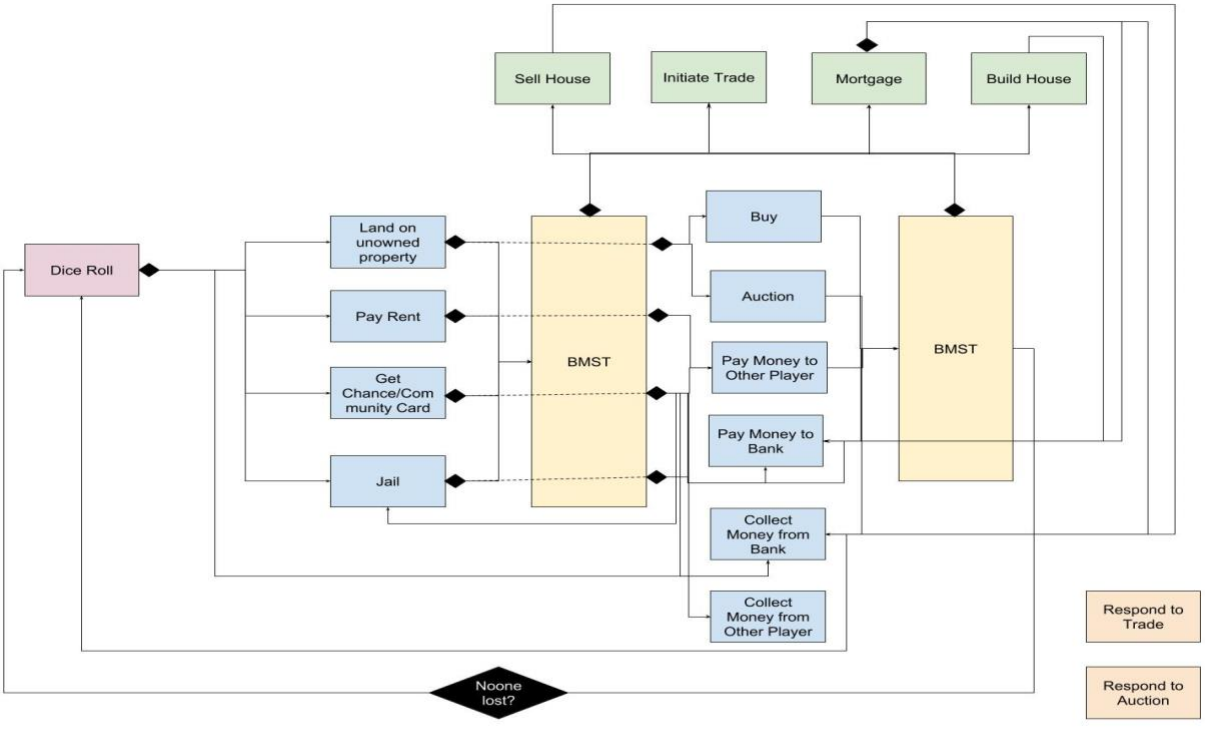


Figure 1: Flowchart of the Adjudicator Program

2.2 Code

The adjudicator passes a Game State around to each of the players. Every input/output occurs to modify the current game state. After validation checks, if the next game state is valid, the current game state is updated and both agents are notified of the new state. This process continues for a fixed number of turns (100 in our rules case) or if a player bankrupts earlier. At the end of 100 turns, the player having a higher total asset (cash, properties and house value minus the debt owed to the bank) wins.

Every turn is defined by a dice roll, followed by one of the following change in positions of the current player:

1. Land on unowned property
2. Land on other player's property
3. Land on self-owned property
4. Land on jail
5. Land on Chance/community card
6. Land on a free parking/safe spot (e.g.: Go)

This change in position triggers a BMST for the players, which we will define later. Following BMST, an appropriate response is given out by the agents, which again if valid, changes the current state of the game. A change in game state calls for an opportunity for agents to BMST again. At this step, a turn ends.

BMST is defined by 4 parts - Building a House, Mortgage, Sell House, Initiate Trade. Each of these functions might evoke a response from the other player - which the adjudicator has been built to take care of.

This also implies that the agents are expected to reply back with a particular response if such a decision is expected from the agent. Lack of a valid appropriate response, simply causes the adjudicator to void the transaction.

At any opportunity to buy an unowned property, the agent may choose to auction it. In such a situation, the adjudicator initiates a blind auction between the two parties. Both the agents reply back with their best price and the one with the higher price wins. In case of a tie, the other player is given advantage over the current player.

In case a trade is called for by the other player, the current player is expected to respond with a yes or no. The adjudicator then changes the game state or moves on with the game.

With the help of the above diagram, we can understand the order in which the adjudicator calls its functions to keep the game rolling according to the rules.

Our adjudicator saves the list of dice rolls in the order which they appear. This is to swap the players and replay the same game, thus giving the beginner's advantage to the other player.

2.3 Test Cases

This adjudicator has been tested to perform as expected on the following test cases:

- Buy property routine (Player 1 decides to buy this property it landed on):
    - Is property allocated to the player bought?
      (Check the change in the game state)
    - Is money deducted from his kitty?
      (Check the change in the game state)
- Buy property routine (Player 1 decides to auction the property):
    - Is auction evoked?
- Auction : Same amount was auctioned (i.e. a tie)
    - Was other player allotted the property?
      (Check the change in the game state)
- Auction : Both players send negative values
    - Neither party is interested in buying, so move on.
      (Check that no change occurred in the game state)

- Jail Decision: Player 1 paid 50 USD to get out of jail
    - Was money deducted?
    - Did he roll for his turn?
- Jail Decision: Player uses Get out of Jail Free card
    - Player's card is used, thus changing the game state's cards owned by the player.
- Jail Decision: Player throws doubles
    - Was 50 not deducted from the player's account?
    - Did player move the double turn?
- Jail Decision: Player doesn't throw doubles
    - No change in Game State
    - Player to remain in jail for next 1-2 turns depending on his history
- Pay Rent: Player lands on the property which other player owns
    - Money deducted from Player 1
    - Money added to Player 2
    - Is the money added equal to money deducted?
- Building Hotel
    - Were 4 houses added to houses remaining with Bank?
    - Was money deducted from player building the hotel?
    - Was a hotel built on the property in question?
- Building house
    - Was the number of houses requested deducted from the state?
    - Was the money deducted from player's account?
    - Does the player own all properties of same colour?
    - Is the state of houses balanced?
- Selling House
    - Is half the amount refunded?
    - Number of houses in "Available houses" increases by 1
    - Is the state of houses balanced?

We understand that these test cases are not exhaustive and a lot more need to be fulfilled before this adjudicator is fully functional. We aim to keep building on top of these cases so as to perfect this adjudicator.

As we keep building the agent, we enrich our comprehension of the game state and add to this list of test cases, thus making it a work in progress.

3 Agent – Current Progress

3.1 Brief Description

Before we could create an agent that would make smarter decisions based on heuristics using Machine Learning models, we decided to create a baseline agent that we could compare our sophisticated agent to. This baseline agent takes decisions based on zero to minimal intelligence, and this was achieved in two iterations as follows.

3.1.1 Phase1: Dumb Agent

For starters, we programmed a dumb agent that would only buy properties and is not interested in anything else. Each function call has fixed return values and did not change depending on the current state. The main motive to build this agent was to collect preliminary data to understand how the game play functioned in a game of 100 turns.

3.1.2 Phase 2: Fixed Policy Agent

Even before we implemented the dumb agent, our intuition made it evident that this agent cannot be used as a baseline model and we had to provide some rudimentary intelligence to our baseline agent. From the data collected, we

gathered statistics and studied them, in order to make some improvements for the decisions our new baseline agent should be making. It yielded us with interesting results based on which we can continue iterating over the agent to develop more sophisticated rules for playing the game in the latter half of this project.

3.2 Implementation Details

3.2.1 Phase1: Dumb Agent

- getBMSTDecision: Always return None.
    - Since this is a dumb agent, it would never consider trading as an option.
- respondTrade: Always return False.
    - Since this is a dumb agent, it would never consider trading as an option.
- buyProperty: Always return True.
    - The intuitive idea behind this was, a player cannot finish their initial cash reserve in one round and after passing go, some of the cash reserve would be replenished.
    - We wanted to gauge the number of turns a game would last if the player kept on buying properties blindly. Turns out, it's an average of 67 turns.
- auctionProperty: Always return 0. Since, we always return True for buying properties, we can be sure that this function would never be called.
- jailDecision: Always return "R". This would make sure that we have the least chance on spending money in Jail.

3.2.2 Phase 2: Fixed Policy Agent

- getBMSTDecision: Always return None.
    - Since this is our first and only baseline model and it'll be competing with itself through the adjudicator, as of now there is no way of taking advantage of 'Trade' option.
    - We did not have enough metrics regarding the properties that can get accumulated by an agent in one game, there was no good enough strategy for the 'Buy' option.
    - Since there is no 'Buy' option, there is no need for the sell option as well.
    - As for the 'Mortgage', blindly buying property resulted in the game going on for 67 turns on an average. The idea was to implement smart decisions while buying the property and see if this count could be increased. The idea would be to avoid mortgaging a property for as long as possible or only mortgage when left with no other option.
- respondTrade: Always return False.
    - As this is our only baseline bot, it will be competing with itself and will never end up initiating a Trade, so this function would never be called.
- buyProperty: Return True if we have enough money.
    - The average number of turns to go around the board of Monopoly is 6-8 considering no jumping to squares and go to jail.
    - Average debt accumulated over each dice roll by a player would range from $25-30. Therefore, keeping a cash reserve of around $240 is the most suitable option. Since this calculation does not include Go To Jail, we added the $50 and kept the minimum sum of money to be $300 for buying a property.
- auctionProperty: Return the value of the property if the agent has the more than $300 money, otherwise return 0.
    - This logic is flawed technically, but will serve its purpose for a good enough baseline.
- jailDecision: Instead of always returning 'R', from the data generated by our preliminary agent, the average turn number by which all the properties have been sold is around 35. We used this to make our jail decisions, if the current turn number is less than 35, and the agent has a Get Out of Jail Free Card in the, use it to get out of jail immediately otherwise pay to get out immediately. If the number of turns have exceeded 35, then roll the dice for the first two times, and use the card if the agent has one in the third turn otherwise roll the third time as well.
    - The idea is that, if all the properties have been sold, intuitively, it's profitable to stay in Jail.

3.3 Key Observations

We ran 1000 simulations using the adjudicator and baseline agents as player 1 and player 2 to figure out if there were any discoverable patterns that we could find which can be used to further improve the heuristics used by the bots to make decisions. We observed the following:

- The most striking observation that we made with the baseline bot was that even though the agents were making decision based on smart policies, Player 1 had a 65% chance of winning the game.
- The game can usually be decided in around 35 turns which is the number of turns by which almost all the properties have been bought through the board.
- The average debt per turn owed by each player was almost equal for the initial 15 - 20 turns but post that the difference became significant as the game progressed. This is shown in Fig. 2 which aids to the observation from the two points above.
- As seen in Fig. 3, the cash held by each player reaches a saturation point in the game around 35 turns, after which the capital held by each player gradually increases. But the difference in expenditure is different for both the players and shows a different rate of increasing for both the players. On an average, the player 1 would be spending more on buying properties and even though spends more, ends up reaping the benefits as the game progresses.
- We also evaluated the number of times a property was bought by each player over the 1000 game simulation (Refer Fig. 4). This tells us that, Player 1, who had an early bird advantage was able to collect more property in his kitty than the second player. However, there exists an interesting anomaly like Property Number 39.
    - A study of this graph will allow us to have a different game play strategy (especially trading) when the agent plays as Player 1 as opposed to as Player 2.
- Fig. 5 evaluates, the number of games in which a player was able to buy a whole group. Buying a group decides the ability of building a house on an owned property. This is necessary in deciding the heuristics for trade of properties.

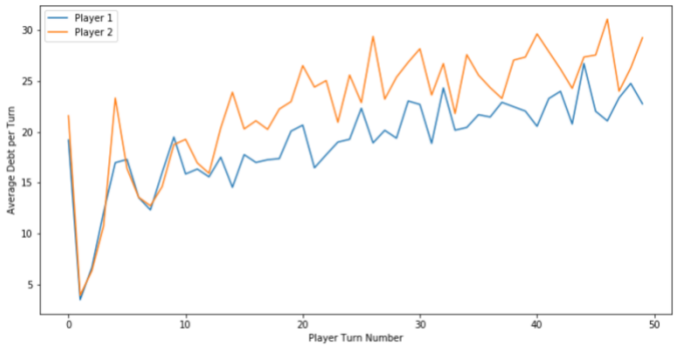


Figure 2: Average Debt Per Turn

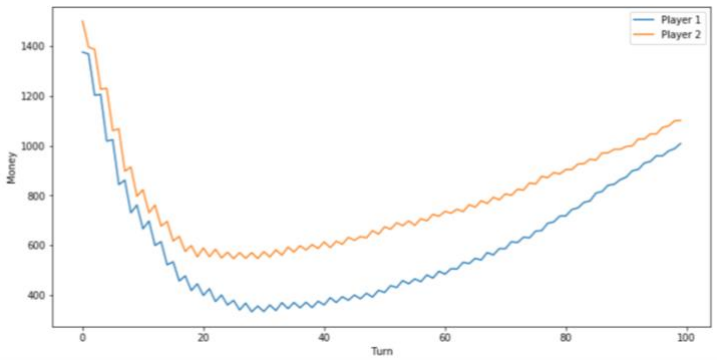


Figure 3: Cash reserve as a function of Turns

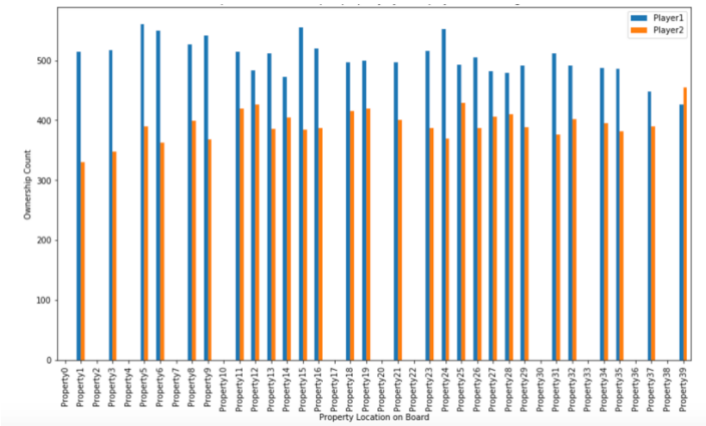


Figure 4: Ownership of Property for each player for 1000 games

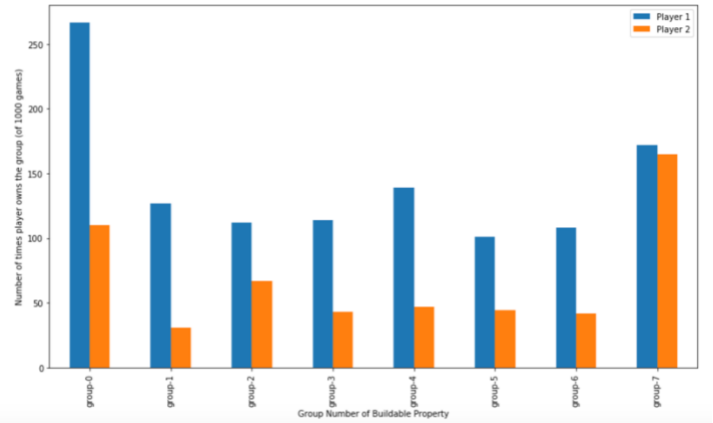
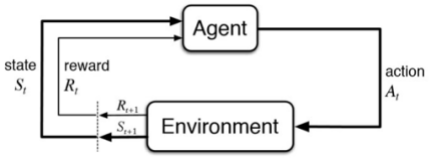


Figure 5: Number of groups of property owned for 1000 games

4 Agent – Future Work

4.1 Introduction

As we have already tried our baseline model above, we somehow need to use the data generated to improve our agent's decision making for better chances of winning the board game. This is where Machine Learning comes into picture, Reinforcement Learning (a popular ML paradigm) which relies on an agent interacting with an environment and learning through trial and error to maximize the cumulative sum of rewards received by it.



RL requires a basic framework called Markov Decision Process (MDP) where the agent is capable of playing and learning winning strategies. And the key elements which helps us convert Monopoly into MDP are as follows :

- *Environment* - The board - the squares, the property cards, chance/community cards, etc.
- *State (S_t)* - Any objects in the environment which basically gets affected by any actions performed on it:
    - Player's coverage on the board (measured by properties owned of different groups)
    - Player's current position
    - Player's finances (properties & total value)
- *Reward (r_{t+1})* - A feedback from the environment. Here, assigning a positive or a negative value for the properties, houses, hotels and cash in hand.
- *Policy* - A function which defines the probability of taking a particular action 'a' on State 'S' at time 't'
- *Value ($Q_{s,a}$)*- Future reward that an agent would receive by taking an action in a particular state.

4.2 Detailed Heuristics

There are some RL models which can be applicable on Monopoly i.e. SARSA (State-Action-Reward-State-Action) Learning & Q-Learning which follows the below equation at any given step/time 't'

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max Q(s_{t+1}, a') - Q(s_t, a_t)).$$

State Space (S_t)- This can be a dictionary of 20+1+2 keys which are - coverage of each group * 2, position, finances (properties and money/value) respectively. Since, every field has its own range and value, to make it all comparable, we will normalize it and bound between [0,1]. An example normalizing would be finances where:

- Properties can be represented as fraction of a player's properties owned over sum of properties owned by both the players i.e. p1 / (p1 + p2)
- Money/Asset value can be stored as it's sigmoid value i.e. money / (1 + |money|)

Action Space (a_t) - Action set applicable on the Environment where each action is stored as vector where each elements represent an action for each of the 10 colour-groups of the game from the following:

- Spend: Consists of buyProperty, buyHouse, buyHotel, unMortgage, Auction
- Sell: Consists of breakHotel, breakHouse, mortgage
- Trade: Special action
- Do nothing

Reward model (r_{t+1}) - The reward signal of the environment is designed so that its output is bounded, provides discriminant ability between positive and negative pairs of states-actions and considers most of the information available to the players. Therefore, a sigmoid function as described previously, is considered suitable since its output values are bounded in [-1,1]. The model can defined as follows:

1. +1 for each property owned by player and -1 for each property owned by others
2. +1 for each house built by the player and -1 for each house built by others
3. +(1-10) for each group (prioritised) owned by player and -(1-10) owned by others
4. Normalize the above sum and bound it by using sigmoid
5. Add a factor of relative money in hand

Rest other parameters in the above equation can be found out by varying the coefficients and making the model learn and finding out the correct set of values which maximises the Q-value in the end.

There are multiple possible models for RL and we intend to implement at least 2 of SARSA, Q-learning, Double SARSA and Double Q-learning in order to find the best fit for the Agent to maximise the chance of winning a game of 100 turns!