Monopoly Playing Program

1 Problem Definition

This project report presents 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.

2 Influential Literature Review

2.1 Generating interesting Monopoly boards from open data, Marie Gustafsson Friberger et al. [1]

This paper explores the possibility of working with data outside a monopoly board game and transforming it into a board game.

2.2 Luck, Logic and White Lies, *Jorg Bewersdorff [3]*

In his book *Luck, Logic and White Lies* Jörg Bewersdorff explores the game of monopoly using Markov Chains defining each square as a state.

2.3 Breakdown of Monopoly - The Game [4]

In her book, "The Indisputable Existence of Santa Claus", Dr Hannah Fry builds on the idea of expressing monopoly sites as Markov Chains and to create a new ranking of different sites, she uses ROI as a metric and calculates how many opponent turns would each site take to return the full investment and start making profit.

2.4 Multi-Agent Reinforcement Learning: A Survey, Lucian Busoniu et al. [2]

The authors have conducted a survey on different Multi-Agent Reinforcement Learning models in this publication. We studied this paper closely and choose the Q-learning method to build the agent.

2.5 Learning to play Monopoly: A Reinforcement Learning approach [13]

This paper allowed us to build the base for building a Reinforcement Learning model for our agent. We defined the action space, state space and reward model following the guidelines of this paper.

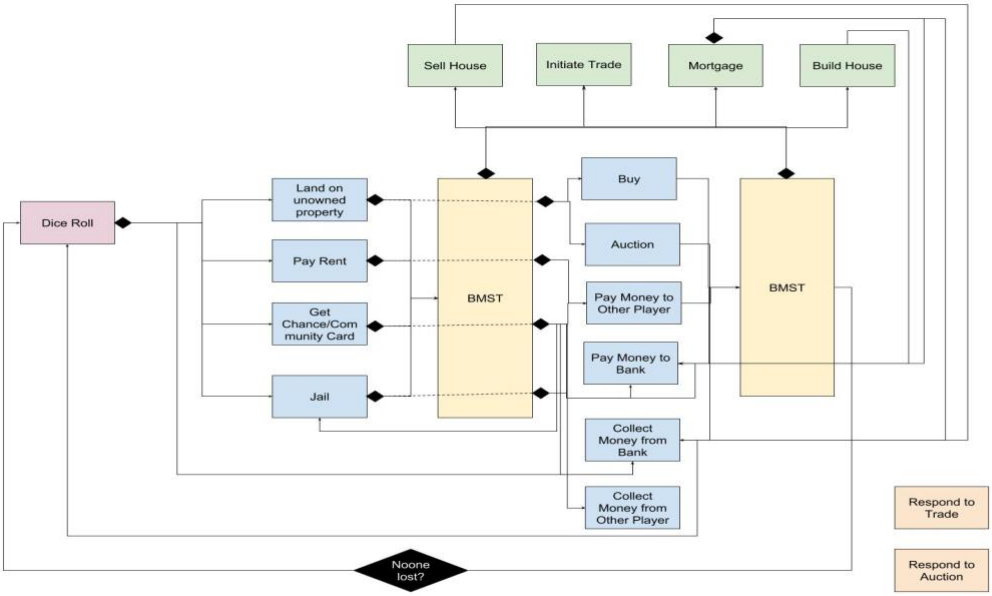
3 Building the Adjudicator

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 built to serve this purpose.

3.1 Design and Architecture

Our 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:



3.2 Implementation

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 (eg: 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.

3.3 Test Cases

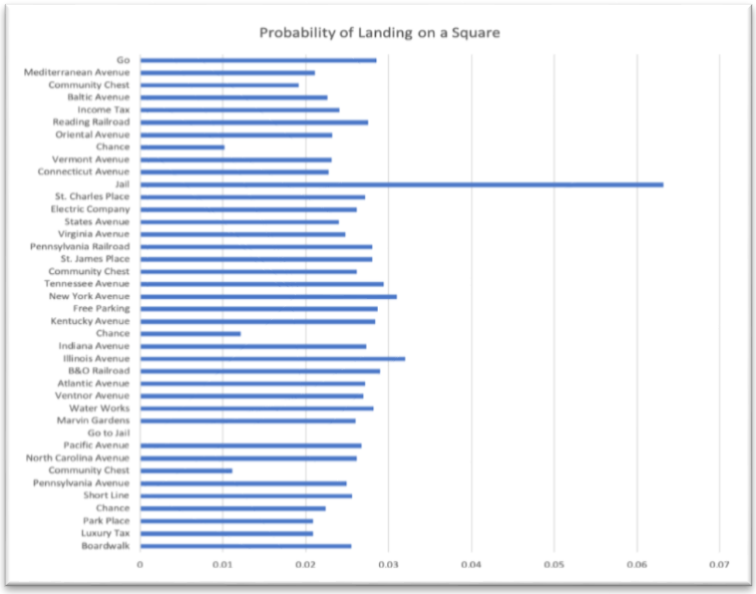
This adjudicator has been tested to perform as expected on the following test cases:

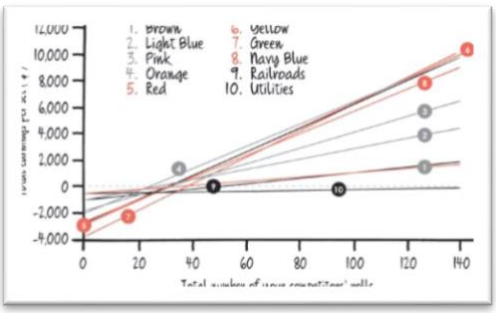
- Buy property routine (Player 1 decides to buy this property it landed on):
    - Is property allocated to the player bought?
      (Check the change in the game state)
    - Is money deducted from his kitty?
      (Check the change in the game state)
- Buy property routine (Player 1 decides to auction 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 color?
    - 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?

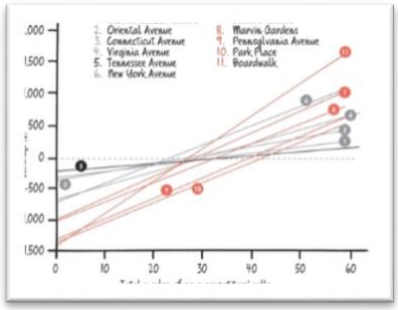
4 Initial Observations

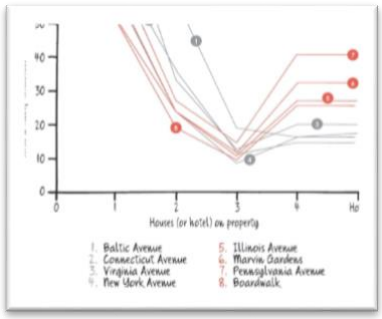
Building on the idea from Section 2.3, we found a simulator [11] which we used to generate the probabilities of landing on each site using a million simulations of a game of 100 rounds each. We ran this simulator and mapped the probability of landing on each square.

While this is good, winning probability is also affected by the houses and hotels that can be built on the site and calculating the ROI on those is equally important. The fastest return on investment according to the calculations is to get 3 houses per set.









5 Agent

5.1 Introduction

We built agents in incremental stages, each smarter than the previous with a significant milestone. They are defined as follows:

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.

Phase 2: Fixed Policy Agent

From the data collected above, 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 as summarized in next section.

Phase 3: Improvised Fixed Policy Agent

Studying some fights between the Dumb Agent and Fixed Policy Agent, we were able to come up with some more concrete statistics that helped us improve the Fixed Policy Agent after the Mid-term report.

Phase 4: Reinforcement Agent

Since the beginning, it was clear that certain functions will perform better with some Learning Algorithm. The final winning agent implements a reward-based strategy to decide its most optimal action.

5.2 Implementation Details

Phase 1: 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 is 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.

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 but will serve its purpose for a good enough Phase 2 Agent.
- **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 behind is that, if all the properties have been sold, intuitively, it's profitable to stay in Jail.

Phase 3: Improved Fixed Policy Agent

To explain the implementation of the 5 functions, we'll first describe a novel variable we introduced to improve the previous agent.

We call a method to evaluate the value of every property after change in state at every turn. Since at every new turn, there may be a change in the ownership of properties, the true price of a property at that instant is

determined by the state at that time. We then move on to create a Priority Queue of all buyable/tradable properties i.e. streets/railroads/utilities ordered by their value.

- **getPropertyValue**: Returns a value for property
    - The value of the property is a function of who has the other properties in the same color set (state):
        - Value of a railroad is 200 times the number of railroads owned. (Possible values are 200, 400, 600 and 800)
        - Value of a utility is 150 times the number of utilities owned. (Possible values are 150 and 300)
        - If ownership of this street property would be the first color in the set, then the value of this property is the true price on the board.
        - If ownership of this street property would be the first color in the set, then the then the value of this property is twice the true price on the board. For the Light Blue block, its $50 more than this price since the ROI for this color set is the highest.
        - If ownership of this street property would be the first color in the set, then the then the value of this property is thrice the true price on the board plus a constant. The value of this constant is determined by the color set.
    - Since not all color sets are equal, based on preliminary work of the previous section, we were able to prioritize color sets based on their landing probability and Return on Investment. The value of the property is also a function of the color set it belongs to.
        - Ordering the color sets based on their maximum ROI, the value of these constants are summarized in the table below:

Light Blue	Orange	Brown	Pink	Red / Yellow	Green
+300	+250	+200	+150	+100	-50

We also create a method to calculate a "threshold cash" for the next turn. This is to ensure we don't overspend in the current turn and run out of cash later, after all mortgaging properties is something we'd like to do only in a dire situation (to avoid losing potential rents and fees on the mortgage).

- **getThresholdCash**: Returns an upper approximation of the cash required for the next turn
    - This method visits every block in the next 2 (min dice roll value) to 12 (maximum dice roll value) and gathers the potentially most cash-intensive block. A chance/community block on the way implies at least $200 must be held back to spare for the next turn.
    - In order to leave a buffer, 1.5 times of this value is returned.

Now, we will refer to the above priority queue, property values and threshold cash to explain the actions of the agent.

- **getBMSTDecision**: Tries to make a decision based on certain conditions.
    - Build Houses: If 5 times the threshold cash is available, then the agent builds "1" house on the property in the top of the Priority Queue.
    - Mortgage: If the agent is in this situation, then the agent already knows how much cash it requires to come out of the turn debt free. This function mortgages the lowest priority property owned by the agent which matches the cash required value.
    - Sell Houses:The agent considers this situation only when houses are built on all properties and it needs to raise liquid cash in the turn to succeed. Again, it refers back to the priority queue and breaks the houses (maintaining the balance) for the lowest required amount.
    - Trade Offer: Any trade offered by this agent occurs in 5 steps. Since one phase of BMST allows only one trade, for a successful trade, this agent might take a maximum of 5 BMSTs to settle.
        - Turn 1: The agent tries to ask for the highest valued property for half its current price (as defined by the getPropertyValue function above).
        - Turn 2: If this trade was not accepted, the agent tries to make a better deal, ask for the same property for 0.6 times its current value.
        - Turn 3: If this trade was still not accepted, the agent tries to make the best possible deal, ask for the same property for 0.75 times its current value.
        - Turn 4: If the previous was still not accepted, the agent picks the lowest valued property it has, and tries to get twice the cash from the other agent for it.
        - Turn 5: Finally, if the other agent is still smart enough to detect this shrewd move, our agent tries one last time to negotiate for 1.5 times the property value and gain that cash instead.
- **respondTrade**: Tries to make a decision based on certain conditions.

    - The agent calculated the net gain/loss it would take if the trade was accepted based on current property values.
    - If the agent would be in a loss, it rejects the trade.
    - If the agent would be in a profit but the threshold cash is not retained after the trade, it rejects the trade.
    - In other cases, it accepts the trade.
- **buyProperty**: If the agent would retain the threshold cash after the trade, it buys the property.
- **auctionProperty**:
    - Our agent checks the cash owned by the other agent. If the other agent raised an auction due to a lack of money on his end, our agent simply bids a dollar higher than what the other agent could afford.
    - In other cases, the agent refers to the current value of the property and makes a bid based on retaining the threshold cash value.
- **jailDecision**:
    - The agent chooses to roll if number of turns elapsed is greater than 35. Based on preliminary results described above, 35 was the turn number when it might be more profitable to stay in jail than to pay and get out.
    - Else the agent uses a card (if available) or pays to get out (again, after retaining the threshold cash).

Phase 4: Reinforcement Learning Agent

Since we have earlier implemented variations of fixed policy agents, we now wanted an agent which used some Reinforcement Learning methods to smartly make the decisions at every step in the game. To start with an RL agent, with Q-Learning as our 1st implementation, thus targeting[12] the first and primary action handled by the bot to progress in the game i.e. buying a Property. We then added a variation to the agent by giving an Epsilon factor thus making it an ε-Learning agent. The ε is nothing but with that fraction amount of probability, we choose our actions from the Action space randomly and the remaining fraction of times we use Q-Learning. As we used Python for writing our bot, we had a few popular frameworks to choose upon such as TensorFlow, Keras, Caffe, Pytorch, Theano, etc and went ahead by using the 'Keras' framework. The basic Deep Learning model used in this framework was 'Sequential' model.

Coming to the Q-Learning concept, as we had mentioned before in our previous report that it has 3-4 parameters such as States, Actions, Rewards, etc, the implementation of those parameters are as follows:

- **Reward function**:
    - +1 for each property owned by player and -1 for each property owned by others
    - +1 for each house built by the player and -1 for each house built by others
    - +(1-10) for each group (prioritised) owned by player and -(1-10) owned by others
    - Normalize the above sum and bound it by using sigmoid
    - Add a factor of relative money in hand
- **Action Space**:
    - For each Actions executed by the Agent, it has its own Action Space which need not be numeric, hence we number those possible actions and then pass those values in the Neural Network to train.
    - Suppose, getOutofJail has 3 possible actions and we number them by 0, 1 & 2 i.e.
        - Jail Free Card
        - Pay $50
        - Roll the dice
- **State Space**:
    - The state we maintain in the Adjudicator has all possible information in a particular instance but not all of them affects the Q-Learning or the Decisions
    - So we extract the relevant information and convert it into the following RL State space:
        - Player's coverage on the board (measured by properties owned of different groups)
        - Player's current position
        - Player's finances (properties & total value)

We hence pass these 3 parameters to the Keras Framework and build the Q-Values for each actions for a particular State and then choose the Action with max Q-Value to be executed at that particular instant. Below is the code-snippet for the parts of the Framework code in the Agent:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import RMSprop
adam = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=None, decay=0.0, amsgrad=False)
rmsprop = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None,
decay=0.0)
task_nn = Sequential()
task_nn.add(Dense(128, input_dim=rl_state_size))
task_nn.add(Activation('relu'))
task_nn.add(Dense(2))
task_nn.add(Activation('linear'))
task_nn.compile(optimizer=adam, loss='mse')
task_nn.fit(dbstate.reshape(bsize, rl_state_size), y, batch_size =
batch_size_var, epochs = epoc_count, verbose = verbose_val)
```

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

```
for i in range(bsize):
     if dataReward[i] == 0:
          y[i][dataChoice[i]] = dataReward[i] + gamma*maxQ[i]
     else:
          y[i][dataChoice[i]] = dataReward[i]
```

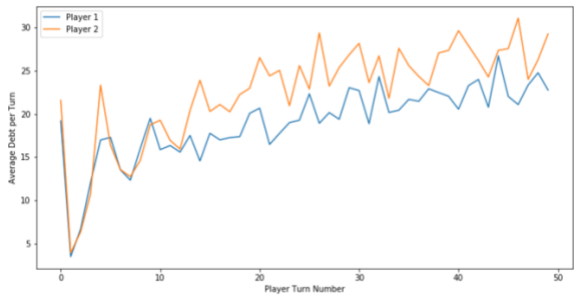
The above parts of code is required by every action function in the Agent file with varying parameters such as the states, action spaces and the rewards. Below are the action function where the RL has been applied:

- **getBMSTDecision**
  - buyHouse:
    - Action space includes number of houses to build at a time with constraints of difference of houses between to properties of same group not exceeding by 1.
    - Each possible action is mapped to a number and then given in training.
  - mortgage: This in particular doesn't require RL because there is only one action.
  - sellHouse: Similar to buy house, it again depends on number houses to be sold at a time meeting the same constraints of difference not exceeding 1.
  - offerTrade: This is the most complicated action to implement RL in it.
- **respondTrade**: This has only 2 options in Action space and that's either to Accept the trade or to Reject. Unlike the counterpart i.e., offerTrade, this one's easy to implement
- **buyProperty**:
  - Although there are already proven statistics of always buying a property if you land on it, implementing RL proves these statistics that on prolonged learning, it slowly converts to 99.99% win rate compared to any random/fixed poicy on the 'buyProperty' function.
  - Again this has only 2 options to make a decision from that is either to BUY or NOT BUY.
- **jailDecision**: This has either 2 or 3 options to decide from based on whether the player owns a Jail Free card or not. The possible actions are
  - 0 - Roll the die
  - 1 - Pay $50
  - 2 - Jail free Card
- **auctionProperty**: This can be implemented in various ways based on how you want to propose an auction price
  - It can vary based on every 1 unit -> 400 options (max property value) or vary on every 50 units -> 8 options
  - The lower the unit value, the better the learning and accurate will be the right auction amount

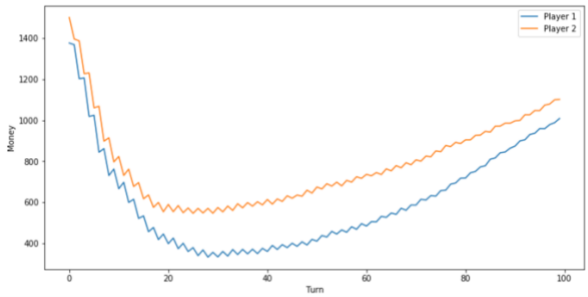
6 Observations

We ran 1000 simulations using the adjudicator and Agents from Phase 1 and Phase 2. We observed the following:

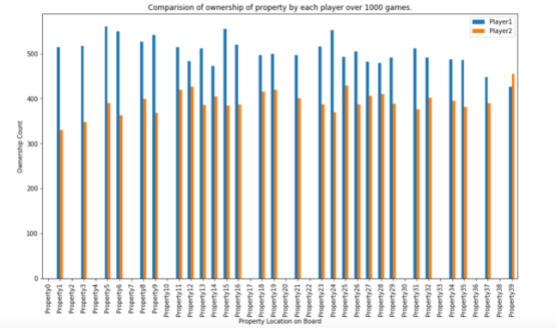
- The most striking observation that we made with the Agent 1 and Agent 2 was that even though the agents were making decision based on smart policies, Player 1 had a 65% chance of winning the game.
- The game can usually be decided in around 35 turns which is the number of turns by which almost all 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 1. This further aids to the observation from the two points above.



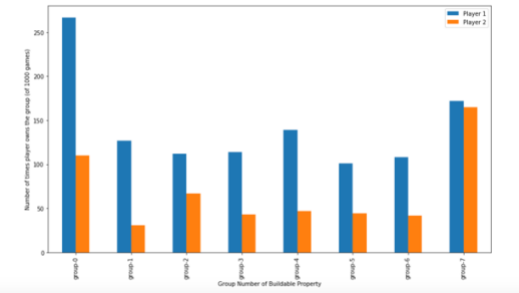
- The cash held by each player reaches a saturation point in the game around 35 turns, after which the capital held by each player gradually increases. But the difference in expenditure is different for both the players and shows a different rate of increasing for both the players. On an average, the 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. 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.

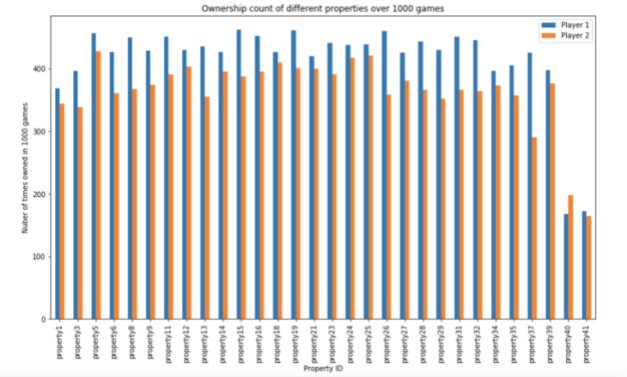


- Next figure evaluates, the number of games in which a player was able to buy a whole group. Buying a group decides the ability of building a house on an owned property. This is necessary in deciding the heuristics for trade of properties.

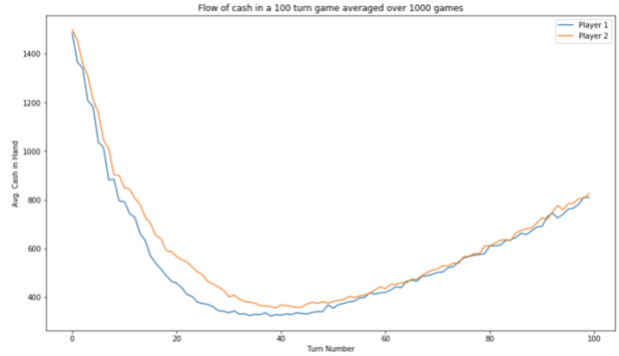


We ran 1000 simulations again using the final selected adjudicator and Agents from Phase 2 and Phase 3. We observed the following:

- Referring to the figure below, Player 1 is Agent from Phase 2 : Fixed Policy Agent while Player 2 is Agent from Phase 3: Improved Fixed Policy Agent. A quick glance at the graph tells us that Player 1 owned more properties more number of times than Player 2. This is because of the first player advantage. Despite this, advantage, our results showed that Agent 2 won 60% of the times. Clearly, a better strategy yields better results and not all is to luck!

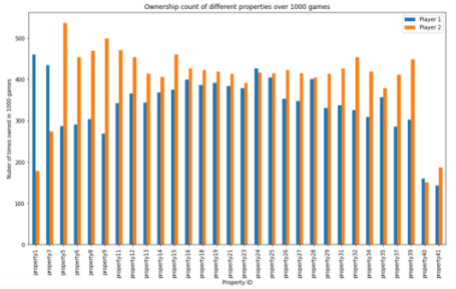


- Between Phase 2 and Phase 3 agents, the difference in cash flow was not significant. Studying the average cash in hand at every turn, we realize that overspending done by Phase 2 Agent causes it to lose. An important take away lesson was that it is important to maintain a good liquid cash flow through the game.



We ran 1000 simulations again using the final selected adjudicator and Agents from Phase 2 and Phase 4. We observed the following:

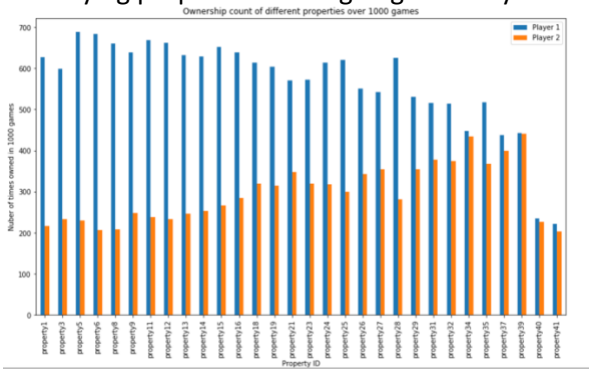
- The observation from the graph is that even though Phase 2 owns properties more no. of times than Phase 4 among 1000 games, still Phase 4 (RL Agent) has the win rate of 60%. Which says that though buying property is good but balancing the liquid cash along with it is also necessary in the long run.



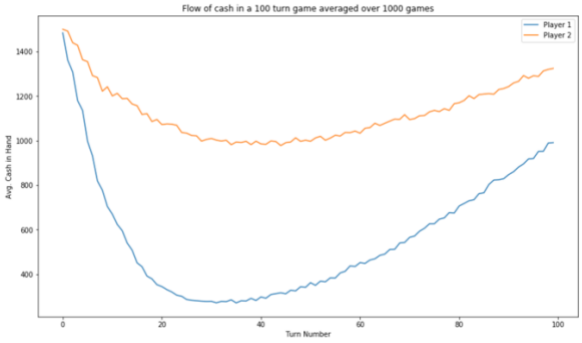
- As we mentioned in the previous graph, that maintaining the liquid cash is also important which is thus proved by this graph above. Phase 4 has been having liquid cash more than Phase 2 at almost every instant thus having upper hand in the win rate. We also observe that both follow the same pattern of decreasing cash flow and reaching a saturation point (buying properties) where almost all the properties are bought and then is the point where rent, trade, mortgage starts showing their effect thus increasing their cash flow gradually.

We ran 1000 simulations again using the final selected adjudicator and Agents from Phase 1 and Phase 4. We observed the following:

- The observation from the below graph is for the Phase 1 bot and the Phase 4 bot over 1000 games. The Phase 4 (RL Agent) has a win rate of around 65% in the games. This was expected since the RL agent is much more sophisticated that the Phase 1 bot. The graph is just to put emphasis on the point that buying properties is crucial in the game. The graph just says that, maintaining liquid cash is good but holding on to too much by not buying properties is not going to lead you to winning the game.



- As we can observe from the below graph that although Phase 1 (Dumb bot) have been maintaining the liquid cash way more than Phase 4 (RL Agent) at any given instant in the game, still Phase 4 wins by 65% win rate which again proves our point that having too many properties or liquid cash is not the right strategy but finding the perfect balance in number of properties and the liquid cash which is best strategy. And we can also observe that if game was run for more tha 100 turns then Phase 4 starts earning the cash with higher rate compared to Phase 1 and would eventually cross the it at certain no. of turns.



Finally a run between each Phase Agent, yielded the following win percentages for 1000 runs each:

	Phase 1	**Phase 2**	**Phase 3**	**Phase 4**
Phase 1	*0*	*60 %*	*65 %*	*65 %*
Phase 2	-	*0*	*60 %*	*55 %*
Phase 3	-	-	*0*	*53 %*
Phase 4	-	-	-	*0*

7 References

1. Generating interesting Monopoly boards from open data, Marie Gustafsson Friberger et al., 2012 IEEE Conference on Computational Intelligence and Games [1]
2. Bewersdoff, Jorg, *Luck, Logic and white lies,* A K Peters, Wesley Massachusetts [2]
3. Monte-Carlo Tree Search: A New Framework for Game AI, Guillaume Chaslot et al., Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference [3]
4. Evolutionary game theory and multi-agent reinforcement learning, Karl Tuyls and Ann Nowe, The Knowledge Engineering Review [4]
5. Multi-Agent Reinforcement Learning: A Survey, Lucian Busoniu et al., 2006 9th International Conference on Control, Automation, Robotics and Vision [5]
6. Here's how to win at Monopoly, according to math experts, Hannah Fry and Thomas Oléron [6]
7. Hacking Monopoly! [7]
8. Computer Game Programming II: AI, University of Southern Denmark [8]
9. Monopoly Board Frequencies and Economics, data set [9]
10. Monopoly Simulations, koaning.io [10]
11. Reference code for Monopoly frequencies [11]
12. Reference code for basic Reinforcement Learning Agent [12]
13. Learning to play Monopoly: A Reinforcement Learning approach [13]
14. Framework Keras [14]