Artificial Intelligence Project Report

Group 20

May - Aug 2022

Project: SplendorAI

Victoria Elizabeth Yong Shu Qi	|	1004455
Phang Heng Yan Gabriel	|	1003466
Daniel Low Yu Hian	|	1004372
Lai Yan Hong	|	1001592

1 Problem Description

(fig. 1: Splendor Board Game Set)

Splendor is a board game of token-collecting and card development, which can be played with 2-4 players. In this project, we attempt to create an AI that is able to understand the goal of the game and play substantially well against humans.

The goal of the game is for the player to be the first to reach 15 prestige points. Prestige points are earned by collecting cards and earning nobles, which both have assigned prestige points value. In order to purchase cards, you need tokens which can be obtained from the bank.

The original game rules include many different actions each player can take during their turn. However, for the scope of this project, we have simplified the rules to form a relaxed problem of the board game Splendor.

2 SplendorAI Game Rules

For this project, the game will be played by 2 players. At each turn, a player can choose to draw tokens from the bank, or use their tokens to buy a card. There are 12 total open cards on the board, which the player can buy if they have the correct tokens. The cards are laid out in rows (see fig. 1), with each row corresponding to a different tier. Each card has a point value from 0 to 5. The first player to reach 15 points wins.

Card

There are 5 different types of cards in Splendor. Each card has a card cost, which is the number of different types of tokens required to buy the card. Each card also belongs to one of 3 tiers, where tier 1 cards are the easiest to obtain (having the lowest costs) but offer the least prestige points. On the other hand, tier 3 cards give the highest number of points but are only obtainable with a large amount of tokens. Cards have a point value between 0 - 5, which varies according to its tier.

In the game, the various card types are recorded in a .csv file (see fig. 2) with the features in order: tier, value, type (colour), green (cost), white(cost), blue(cost), black(cost), red(cost).

```
tier,value,type,green,white,blue,black,red

1,0,1,0,2,1,0,0
1,0,2,0,0,0,1,2
1,0,3,0,1,0,2,0
1,0,4,2,0,0,0,1
1,0,5,1,0,2,0,0

1,0,1,0,0,0,0,3
1,0,2,0,0,3,0,0
1,0,3,0,0,0,3,0
1,0,4,3,0,0,0,0
1,0,5,0,3,0,0,0
```

(fig. 2: Card CSV data)

Noble

Nobles are special types of cards, each with a point value of 3. There are 3 obtainable nobles in every 2-player game. Each noble also has a cost. However, the cost of the nobles only counts the number of cards owned by the player and excludes the tokens they possess. For example, if a noble costs 3 red cards, 4 blue cards and 3 white cards, the player will be awarded with that noble if it is on the board, but having 2 red cards and 1 red token does not count.

At the end of each player's turn, the game checks if that player has the requirements for any of the nobles. If the player fulfils the card requirement, they will be awarded with the noble. Players are only able to obtain a maximum of one noble at the end of their turn, even if they fulfil the requirements for multiple nobles. If a player is able to receive more than 1 noble at the end of the

turn, they will get the first available by index instead of being able to select one as in the original rules. This removes the need for the AI to decide which noble to take.

Similar to the cards, the Nobles' data are stored in a .csv file containing each noble's card requirement (fig. 3)

```
green,white,blue,black,red

4,0,4,0,0
4,0,0,0,4
0,0,0,4,4
0,4,0,4,0
0,4,4,0,0
```

(fig. 3: Noble CSV data)

Player

Each player is able to collect tokens and cards throughout the course of the game. A player can only collect a maximum of 10 tokens of any colour but can collect as many cards as they are able to buy.

A player can take either one of 2 actions (see The Board), take tokens from the bank or buy a card. Each player will also keep a tally of how many of each card type they own. This is the qualifying factor for receiving a Noble and reducing the token-cost of cards. Finally, a player must keep track of the number of points they currently own.

Each player has the following attributes:

id<int>: Player ID
turn_order<int>: Number representing their turn among other players
points<int>: Sum of value of owned cards and nobles.
cards<List<Card>>: List of cards owned
card_counts: List of the number of each card type owned
tokens: List of the number of each token type owned
nobles: List of nobles owned

Bank

There are 5 kinds of tokens in the bank. For our simplified 2-player version of the game, the bank starts with 4 instances of each token type. When a player buys a card, the tokens are deposited back into the bank.

The Board

The board has 3 rows of cards, with each row corresponding to a different tier. Each tier has 4 open cards at any time except if there are insufficient cards of the corresponding tier to fill the slots.

Player Action: Buying Cards

Players are able to purchase open cards. Once purchased, the point value of the card is added to the player's point tally and the number of cards for that specific colour that the player owns will be updated. A card will be drawn from the respective deck to replace the one that was purchased. After all card movement and calculations are completed, the game checks for any nobles that can be earned and finally checks if there is a winner.

Player Action: Taking Tokens

The player can choose to either take 2 of the same token, provided there is at least 4 of that token type in the bank, or single tokens from 3 different types. They can also choose to take less tokens than the allowed number to avoid going over the 10 token limit.

3 Minimax

One of the first considerations when building the AI for Splendor was to use the Minimax algorithm, along with alpha-beta pruning for efficiency. For our simplified game, these are the actions a player can make.

Action	Number of combinations
TAKE_TOKEN	${}^{5}C_{3} + {}^{5}C_{1} = 15$
BUY_CARD	4 cards per row * 3 tiers = 12
Total	15 + 12 = 27

One of the trickiest things to do was to assign value to the current board state. We decided to assign a numeric value to the board state from the perspective of the AI for simplicity, where a positive board value indicates that the AI is winning, while a negative board value indicates the opponent is winning. The values below indicate how much value is assigned to each condition.

```
POINTS_VALUE = 100
T3_CARD_VALUE = 40
T2_CARD_VALUE = 30
T1_CARD_VALUE = 20  # cards must have a value higher than their tokens
GOLD_TOKEN_VALUE = 2 # irrelevant for simplified game
BASE_TOKEN_VALUE = 1
```

In addition, we had to give additional value to the player who is going to be able to buy a card the next turn, in order for the algorithm to choose the path that allows for them to buy a card as soon as possible. How much additional value to assign to this state is calculated with the following formula:

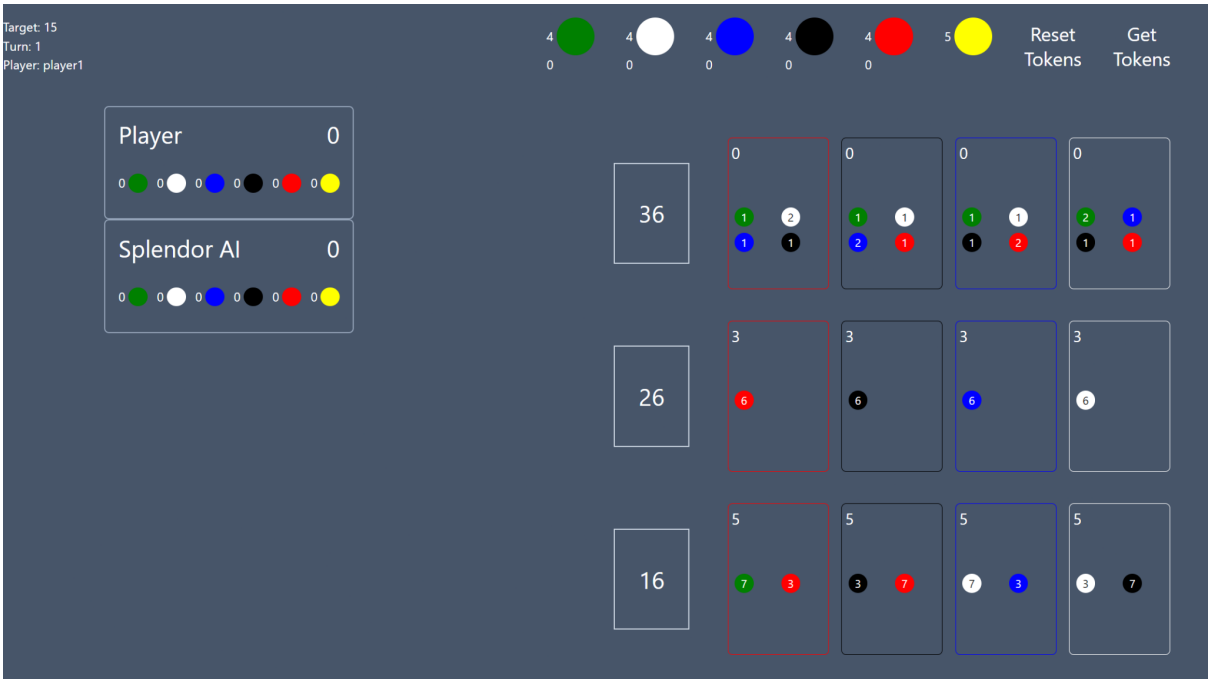
$$value = Tier\ of\ card\ value - Sum\ of\ spent\ tokens - regularizer$$

The regularizer is needed so that the AI actually decides to buy the card instead of holding the tokens.

3.1 Code setup

Run the flask application with flask run within the data folder to start the backend. Run the React Frontend application inside the ui folder with npm start after installing dependencies with npm install.

3.2 GUI gameplay



(Fig 4: GUI of the game running with minimax algorithm)

The player will be able to interact with the AI through a web app, built with React. Once they have made their move, the web app will make a POST call to the backend, which will then be able to run the minimax algorithm to obtain the best move for the AI, given the new board state. The AI then makes that move and returns to the frontend that new state. The game state is always modified in the backend, which contains all game logic.

3.3 Evaluation

From our testing, the optimal depth of the minimax algorithm is 3 layers with our current implementation. Any more would result in the algorithm taking over 5 minutes to execute, which was too long. With 3 layers and a total of 27 actions per move, that results in the AI computing over $27^3 = 19683$ different moves. Alpha-beta pruning helped to reduce the overall number of moves computed significantly. On average, the AI takes 14.13 seconds to compute its next move.

Given enough time, the implementation for the game's logic could be further refined, which could potentially increase the depth. Multithread processing could have also been used to speed up computation time.

4 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a well-known application of Monte-Carlo methods to adversarial search algorithms, often employed in games where heuristic evaluations are difficult to find or branching factors are too large for a traditional minimax search, such as Go. Recently, with the advent of AlphaZero and other game AI programs which combine MCTS with reinforcement learning, there has been renewed interest in the applications of MCTS to game AI problems.

For this project, we implemented a “traditional” MCTS, where heuristic evaluations are determined by random playouts. MCTS involves several phases: a selection phase where promising nodes are selected until we reach a leaf (i.e. unexpanded) node, an expansion phase where new nodes are generated from leaf nodes by creating new nodes for each possible future move, an evaluation phase which consists of a traditional rollout, and backpropagation, which propagates statistical information to the root of the game tree.

4.1 Evaluation

Empirically, with 20 seconds of runtime on an M1 laptop and 10 iterations per rollout, we are able to run roughly 600 iterations of our selection/expansion/evaluation/backpropagation phases. Even so, the algorithm seems to give nonsensical suggestions at times, such as suggesting one to pass their turn even if there are better moves available. To try to combat this, we changed the exploration policy to prioritise buying cards instead of taking tokens, and taking tokens instead of passing one’s turn. Empirically, this change has resulted in faster self-play games, which does seem to indicate an increase in playing strength.

Part of the lack of performance was perhaps due to inefficiencies in performing rollouts, and also the necessity to perform copies of our board state on each new node, which due to our nested object structure was relatively inefficient. In addition, some future work may be to replace our policy and evaluation functions with the heuristics generated by our Deep Q-network as indicated below (we deliberately implemented our evaluation function as a callback so that it would be possible to use better heuristic evaluations).

5 Reinforcement Learning

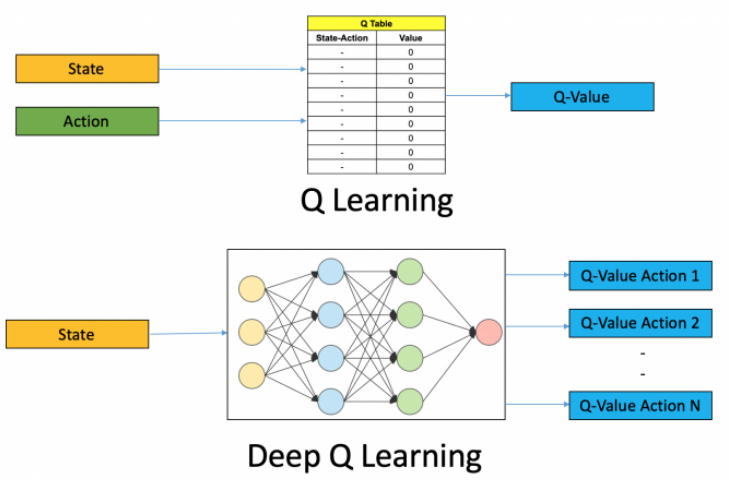
For this project, we attempted to train an RL model using the Deep Q-Learning algorithm.

Q-learning is a model-free, off-policy reinforcement learning that learns the value of an action taken given the current state of the environment. Depending on where the agent is in the environment, it will learn and decide the next best action to be taken. Q-learning finds an optimal policy, maximising the expected value of the total reward over any and all successive steps, starting from the current state (fig. 6). In basic Q-learning, a Q-table is used to iterate through the reward value of actions taken in a particular state. The table is first initialised to zero, and its values are iteratively updated during training.

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

(fig. 6: Q function equation)

In Deep Q-Learning, a neural network approximates the Q-value function, which replaces the traditional Q-table (fig. 7). Deep Q-Learning aims to train a policy that tries to maximise the discounted, cumulative reward. Every Q function for some policy obeys the Bellman equation, and the difference between the two sides of the equality is known as the temporal difference error, δ.



(fig. 7: QL and DQL Architecture Diagrams)

In problems with many different possible states and many permissible actions at each state, the resulting Q-table would be unsustainably large. Learning the optimal values in such a model would take an unreasonably long time and require a lot of disk space to store the information for each state. Furthermore, the memory required to save and update that table would increase as the number of states and actions increases, which adds to the time complexity of the overhead for such an operation. In problems where it is not easy to derive the Q-value of new states from

previously explored states, the deep neural network provides a plausible approximation, enabling the model to learn the reward values of different actions.

5.1 Modified Rules and Inputs

Training an RL model requires specific input and output data, particularly since a Deep Q-Learning network includes a neural network. Thus, it was necessary to rewrite the game rules such that the game system would return data in the ideal structure for deep learning.

The essence of the game rules remained the same. The game handler was modified to return the reward of an action taken, and a boolean variable signalling if the game had ended or not. A simple reward function is used, prioritising actions that result in an increase in player score. The reward values are given as such:

```
+1 if taking tokens
+card_value if buying card
+ 3 if win game
-1 if lose game
```

The main game loop was implemented as following pseudocode:

```
        def playerAction(self, action_index):
# action index is a number between 0 to 23,
# and is the output of the deep q network
reward = 0
if action_index in range(0, 12):
    # 12 ways the player can buy a card
    card = buy_card()
    reward = card.value
    update_player_states()
    update_board_cards()

elif action_index in range(11, 27):
    # 15 ways the player can take a token
    reward = 1
    take_tokens()
    update_player_states()
    update_bank_tokens()

check_noble_visit(current_player) # checks if current player can recieve a noble
winner = check_winner() # checks win condition. returns player if they won.
reward = 3 if check_winner == player0 else -1 # always favour first player
done = True if winner else False
return reward, done
```

SplendorAI's simplified game rules allow a total of 27 actions at each player turn. The input to the model is a tensor of dimension (1, 129) representing the game state (fig. 8). The variables represent the key information describing the type of cards open on the board, the available nobles, both players' state information (points, tokens, etc), and the bank's tokens.

```python
def getState(self):
    # returns 1d list of board states
    # 12 open cards (12, 7), 3 nobles (3, 6), player1 (1, 11), player2 (1, 11), bank (1, 5)
    dims = (12 * 7) + (3 * 6) + (2 * 11) + 5
    data = np.zeros(dims)
    cards_state = np.zeros((12 * 7))
    nobles_state = np.zeros((3 * 6))
    players_state = np.zeros(22)
    bank_state = np.zeros(5)
```

(fig. 8: Function to get game state)

5.2 Single Deep Q-Learning Network

In the Single Deep Q-Learning Network, the model becomes its own adversary by controlling the actions of both player 0 and player 1. The model attempts to choose the best possible action at each player turn, while also attempting to make player 0 win.

Implementation

The basic architecture of the neural network part of the algorithm is a feed-forward neural network, consisting of linear layers with ReLU nonlinearities and a sigmoid output activation function (fig. 9).

```python
def forward(self, x: torch.Tensor):
    x = x.to(self.device)
    x = F.relu(self.linear_1(x))
    x = F.relu(self.linear_2(x))
    x = F.relu(self.linear_3(x))
    x = torch.sigmoid(x)
    return x
```

(fig. 9: Neural network architecture)

Experience replay memory (fig. 10) is used in training the deep Q network (DQN). Recent transitions that the agent observes are stored in the cyclic buffer, ReplayMemory, allowing the data to be reused later. Data is sampled from the log randomly, allowing the transitions that build up each batch to be decorrelated, which greatly stabilises and improves the DQN training procedure.

```python
class ReplayMemory(object):

    def __init__(self, capacity):
        self.Transition = namedtuple('Transition',
                                     ('state', 'action', 'next_state', 'reward'))
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(self.Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

(fig. 10: Experience replay memory code snippet)

In training the model, Huber loss is used (fig. 11). Huber loss is used in robust regression, and is less sensitive to outliers in data as compared to squared error losses. PyTorch's implementation of Huber loss is a pseudo-implementation which approximates the Huber loss function. It combines the best properties of L2 squared loss and L1 absolute loss by being strongly convex when close to the target, and less steep for extreme values.

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r) \in B} \mathcal{L}(\delta)$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

(fig. 11: Huber loss function)

Compared to the original Huber loss function, the differences in the PyTorch's Smooth L1 loss function are:

- As beta -> 0, Smooth L1 loss converges to L1Loss, while HuberLoss converges to a constant 0 loss. When beta is 0, Smooth L1 loss is equivalent to L1 loss.
- As beta -> +∞, Smooth L1 loss converges to a constant 0 loss, while HuberLoss converges to MSELoss.
- For Smooth L1 loss, as beta varies, the L1 segment of the loss has a constant slope of 1. For HuberLoss, the slope of the L1 segment is beta.

The model uses the Root Mean Square Propagation (RMSProp) optimizer, an adaptive optimizer often used in mini-batch gradient descent. In optimising the model, we calculate the Q(s_t, a_t) and V(s_{t+1})=\max_a Q(s_{t+1}, a), and combine them into the loss. Since our DQN is a feed-forward network, the input dimensions are fixed. Hence, in order to calculate the max Q value over all sampled states, stochastic batching is implemented.

The model takes samples of pre-defined batch size, and the sampled states are loaded into a PyTorch DataLoader class using a custom Dataset (fig. 12), and each state is fed into the DQN one at a time. The outputs are then collected and combined into the loss function using a mask.

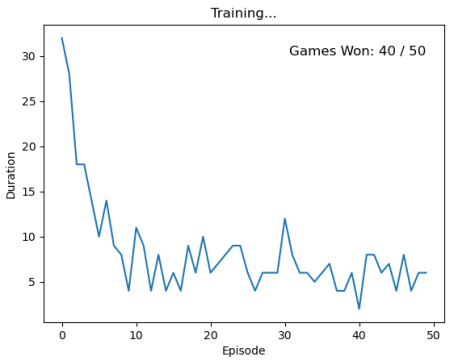
```python
class BatchData(Dataset):
    def __init__(self, data):
        # data is a list of n input states, where n is the batch size
        self.states_data = data

    def __len__(self):
        return len(self.states_data)

    def __getitem__(self, index):
        data = self.states_data[index]
        return data
```

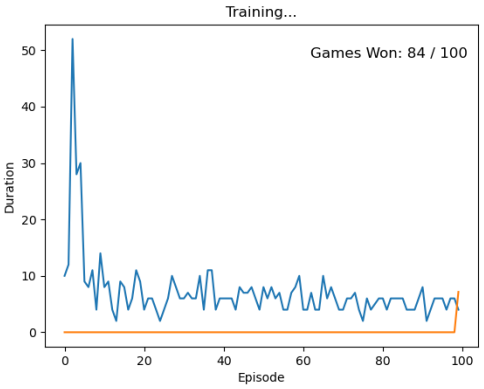
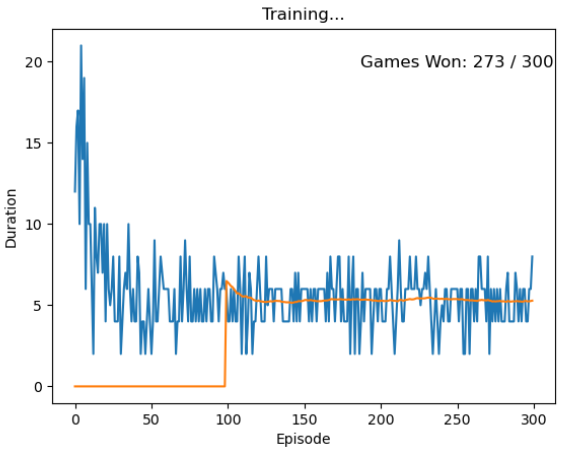
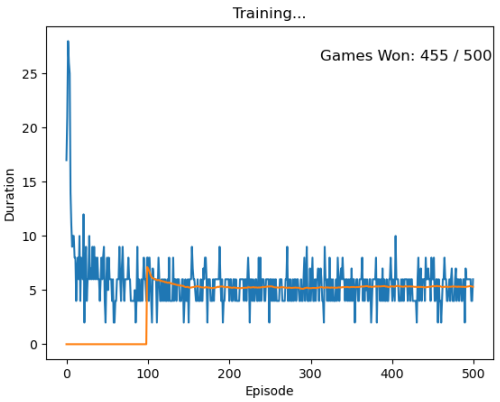
```python
state_dataset = BatchData(batch.state)
state_batch = DataLoader(state_dataset)
action_batch = torch.cat(batch.action)
reward_batch = torch.cat(batch.reward)

# Compute Q(s_t, a) - the model computes Q(s_t), then we select the
# columns of actions taken. These are the actions which would've been taken
# for each batch state according to policy_net
values = torch.empty((128, 27), device=self.device)
for i, batch in enumerate(state_batch):
    values[non_final_mask] = self.policy_net(batch)

state_action_values = values.gather(1, action_batch)
```

(fig. 12: Custom Dataset and Dataloader for optimising model)

Results

The model was trained over 50, 100, 300 and 500 games, with a batch size of 128. The optimizer's gamma was set to 0.999, and the learning rate was set to 0.9 with a 200 decay. The model's replay memory was set to log a maximum of 10, 000 transition states

No. Games Played	Graph	% of Games Won
50		80%

100		84%
300		91%
500		91%

The model's performance appears to gradually improve with an increasing number of games played, up to a maximum of 91% of games won with 300 and 500 games. The results seem to suggest that the model is successful in understanding the rules of the simplified game, and making decisions to win the game.

However, due to the need to modify the game rules to return the desired data and variables for the model to work, the RL model is not compatible with the GUI system, and we are unable to test the model against a player in a real setting.

6 Conclusion

Given that we have used 3 different models to implement the AI, the evaluation of each model is done differently. Overall, the Minimax algorithm has the most practical usage, as it is able to function together with the GUI to produce a playable version of the simplified Splendor.

The RL model seems to show potential, however, without the ability to conduct proper testing, it is difficult to discern if the reward function chosen was suitable, or if it was too simplistic to achieve our goals.

6.1 Future Improvements

Despite having developed and tested 3 different algorithms to solve the game of Splendor, we were unable to come to any satisfying conclusion on which model was the best. This can be attributed to the completely varying evaluation metrics used. Collating and standardising our evaluation functions would help in achieving our goal. One example would be to run all 3 models with the same GUI and game system, and measure their performance in time taken to decide on an action, and the number of games won against the same player.

Other improvements could include:

- Double Q-Learning Network
- Further optimising the game system
- Expanding the problem to encapsulate the full ruleset of Splendor

7 Addendum

The code to our project can be found here:
https://github.com/omnifarter/50.021-Splendor-AI/tree/submission