CS 747 Project

Team:

1. **Abhijith Dimitrov 130020012**
2. **Palak Jain 130050031**
3. **Vaibhav Bhosale 1300500007**

Problem: The task is to generate an optimal agent for the carrom game which is able to capture the entire board in case of single player and the half colour coins in case of two player in as minimum number of moves possible.

Our approach initially was to model the game as an MDP and solving it to obtain the optimal policy. We thought of the following model for the same:

Modelling the Game as a Markov Decision Problem

Motivation: There exists a striking similarity between the two problems- MDP and the carrom game. In MDP, given a state, we need to determine an action that will give maximum reward in the long run, whereas the aim of the game is the same i.e. clear the board in least possible moves.

Model: The state space and action space taken in the entirety of all the possible ones. In a MDP, for a state, when an action is chosen, you can transition to more than one states probabilistically and receive rewards accordingly.

In the current scenario, this translates to choosing an action for a state, and generating the probability distribution based on the value of noise added.

Example: Suppose we are in state s, and choose an action a = (0.5, 90, 1). Since we have noise, the next state would be generated by the modified action a' = (0.5 + noise_x, 90 + noise_angle, 1 + noise_force). The noise can hence be discretized and the next state obtained with probabilities same as that of the noise i.e.

$$Pr(s, a', s') = G(noise\_x, 0.05) * G(noise\_angle, 2) * G(noise\_force, 0.01)$$

Using this model, and sampling all the possible states, apply Policy Iteration method to find the optimal policy and use that in the game.

Strengths:

This model allows to cover the entire state space and action space and helps deduce the optimal action for every state possible.

Drawbacks:

1. Since the state and action space is very large, the time needed to sample is very large.
2. Similarly with large state and action space size, the time required for convergence of the Policy Iteration method is very high.

The fundamental problem with learning using standard techniques is continuous state and action space. Most learning algorithms are designed for discrete finite state-action pairs.
The earlier model discretizes at a very fine level, but it becomes computationally infeasible. Therefore we try to reduce the state and action space using some statistical and geometric properties of the game.

Noise inclusive discretization

Motivation

Statistically the noise added to the action, can be used to model the space around the action as well. Since, the probability of the noise being greater than standard deviation is small, we discretize the action space such that two discrete points are approximately at a distance of twice the standard deviation as described in the model below
We divide the entire state space in discrete blocks as per the coin radius. The board is divided into (board size/2*coin size) along both axes. Our state would then be a vector of size number of blocks consisting of color coding (None, Black, White, Red).

Algorithm

To discretize action space, we take into consideration noise. Our actions can always be deviated by certain amount due to noise. So, for desired position x, our final position can cover the range [x - 0.005, x + 0.005]. Thus, with x and x + 0.01, we can hope to cover all possible x's.
However, the action space would be very large with this level of discretization. By experimentation, we noted that an appropriate angle can compensate for a gap in position.
Thus we chose the action space to balance between covering all possible actions and minimizing the space to increase training speed of our algorithms.
We divide position into 20 segments, angle into 50 and force into 40.

Drawback

The overall state-action space was still huge for any algorithm to converge in reasonable time.

Further, we try to reduce the state space depending upon the geometric properties of the board. Thus we try to extract certain features from the state as in the following model:

Feature Extraction

Motivation
A large state space is a major obstacle for any training algorithm.
Instead of dividing the state space in blocks, we realized that coordinates are merely features of the state. Therefore, instead of describing the state in terms of position of each coin, we extract features of the state based on locality and spatial information.

Model

Define state in term of features extracted from state and perform learning algorithms over it.

Possible features include:

- Nearest distance to pocket of a coin
- Distance from striker line
- Number of coins in proximity i.e cluster formed by coins
- Number of obstructions between coin and pocket
- Position of coin

Besides these we can also reward or punish a coin based on its

- Coin in a region we cannot strike at
- Coin easily pocketable by opponent

Strengths

- Reduces the state space drastically.
- Providing extracted features instead of positions might help in training.

Drawbacks
Feature extraction is non trivial. We might over emphasise certain features while missing out on the important once.

Necessity of learning techniques

A major hindrance in the above techniques was our inability to store Q values for a large state space. However, designing a function that could return Q for given s,a can simplify our problem.

Q-learning is one kind of reinforcement learning method that is similar to dynamic programming and the neural network has a powerful ability to store the values
Also online learning and adaptation are desirable traits for our model.

Our algorithm

1. Initialize the network weights for neural net N.
2. Get current state s.

Repeat until convergence:

3. Generate action space A for the state s. We can either use the entire space or prune it using heuristics.
4. For every action a in action space A :
   Forward propagate for s, a
   Q(s, a) <= N(s, a)
5. Generate probability distribution P from Q(s, .) by passing it through softmax.
6. Pick action a* from P
7. next state s', reward r <= Simulate the game using state s, action a

8. For every action a in A:
   Q(s', a) <= N(s', a)

9. Pick a' = argmax Q(s', a)
10. Get TD-error d = reward + discount*Q(s', a') - Q(s, a*)
11. Update the network weights by backpropogating TD-error

Architecture

We provide as input State x = { x1, ..xn } and action a = {a1, a2, ..ak } to the neural network and output Q(x, ai) for each feature of action.
Input:
{ State + Action } is a 1D vector comprising : NumOfCoins X [x,y] + [position, angle, force]

Its dimension is 19X2 + 3 = 41

Output: is a single node of size 1 that represents Q(s,a).

1. Neural Network to generate Q-value

Hidden **:** Our network has one hidden layer of size 30 units and activation function as tanh.

$$Q(x, a) = \text{Linear1}(\tanh(\text{Linear1}(x, a) + b1)) + b2$$

Reference : http://www.ice.ci.ritsumei.ac.jp/~ruck/CLASSES/INTELISYS/NN-Q.pdf

Results and Analysis

For the entire game, i.e starting from the initial state comprising of all the coins, we ran our model for 50 games, each capped at a max of 500 episode length. We recorded the number of time steps for game to end for 1 player.
However, the number of time steps did not seem to converge. There was a high fluctuation in the result with average around 80-100 steps. The network reached a minimum of 38, however in some cases while training it also overshot beyond 200.
We were not able to train for a large number of games due to resource and time constraints. It is possible that training for a long time might have led to some convergence.

Learning for SubGames

Motivation: Our heuristic model works reasonably well for the initial states when there were clusters of coins. But as the coins get separated out and we try for straight shots, the difference due to noise began having a larger impact, resulting in misses/fouls in quite a few cases. Therefore, we decided to learn for as many such states possible where there is only one coin on the board.
Model: The state space is such that all states are basically single points on the board chosen at random where one coin is placed. We predict an action a' = (x', a', f') using our heuristic model, and limit the action space to certain values around the predicted action a' i.e. position varying in the range [x'-0.25, x'+0.25], angle in the range [a'-30, a'+30] and force in the range [f'-0.1, f'+0.1] with the same discrete interval as in the earlier model. If a coin is pocketed, the model obtains a reward 1, whereas with every timestep for which the coin isn't captured, the model obtains a negative reward, increasing in magnitude with every passing time-step, thus constraining the model to capture the coin in as fewer moves as possible.

Strengths:

1. This model allowed us to improvise our agent in specific poorly performing instances.
2. It gives an optimal action which would enable capture of the coin in minimal number of moves even with noise
3. This can also be used in the cases when there are more than one coins but there exist a coin having an unobstructed path.

Drawbacks:
Since the state space is very large, we couldn't train the model to reach convergence

Analysis:

For a single piece game, we obtain an average of 17.

While for 2 pieces, our average is 30.

Heuristics based Approach

Single Player

We have two broad strategies in capturing coins. The intuition behind it being that when coins are clustered, hitting with a significant force would make them spread out and help us have direct shots at them. This also resulted in significant number of coins being captured on hitting the cluster.

1. Detect Clusters of coins within a certain threshold radius and minimum size of cluster:
    a. Prioritize clusters near pockets using a metric combining cluster size and pocket distance. If pocket distance of cluster is too large, we consider a second metric involving only size.
    b. Aim at the centre of clusters if there is a coin in the coin_radius proximity
    c. Otherwise, aim at the coin nearest to centre. This is so as to avoid a blank hit by striker
    d. Force is computed proportional to size of cluster and combination of striker distance and pocket distance.
    e. The position of striker is determined by iterating over the possible x-coordinates of the striker ranging from 170 to 630 incrementing by a discrete step of 10 units.
    f. Further, we check the projection angle of the vector from the striker location to the cluster center on the vector joining the cluster center to the pocket location, and try to assign the position with minimum projection angle.
    As the projection angle decreases, the coins in the cluster break away in the direction towards the pocket which results in more coins being pocketed.

2. Cut-shots
    a. We aim the striker towards the coin at an angle < 90 degrees to the coin-pocket vector such that the the striker on hitting the coin propels it towards the pocket. The striker position and hence angle w.r.t base is determined by choosing the path with no obstruction and reducing distance covered.
    b. Force is proportional to the angle between coin - pocket and coin - striker and the distance of coin - pocket and striker - pocket
    c. Also, this strategy enables us to counter the issue faced when hitting striker straight along the line and encountering fouls.

Strategy for queen

Case 1 : # coins > 10

We focus on clearing the clusters, which enables us to win a lot of pieces and in several cases even capture the queen

Case II : 10 > # coins > 5

We aim at queen if we find a shot at an angle and subsequently, if we capture the queen, our strategy which aims at pocketing a piece at every turn works efficiently to win a cover.

Case III :

We prioritize capturing the queen by employing greedy strategies and similarly for cover.

Other notes:

1. For the first move, we hit the coin from the middle with force 1 so as to break the cluster and maximise the spread.
2. We ensure that the striker positions we sample from are never blocked by another coin placed there previously.
3. If our cluster size is large enough, we aim at it with maximum force.

Two Player

The major difference from single player is that we are only allowed to pocket coins of color assigned to us. If any coin of opposite color goes in, we get zero reward, hence we have to be more careful while hitting to ensure this doesn't happen.

i) First shot - In case we are the first player, we try to bring coins to our side of board using a rebound shot with sufficient force. It's easier for us to hit coins on our side because the striker-coin + coin-pocket distance is less and hence the noise is less likely to affect the outcome of the shot as long as we control the force.

ii) We then check if there exists an **unobstructed cut-shot** for any of our coins. If there exist multiple such shots then the one with best metric as in single-player case is chosen.
iii) If shot not found above, we look for **clusters** of our coins on the board and attempt to break them (in the process also pocketing coins by aiming the shot such that that it is approximately aligned to line joining cluster center and nearest pocket).

Clusters are implemented in the same way as single player, by calculating number of coins present within threshold radius of each coin. The only difference is that, priority of cluster is now proportional to **(number of our coins - number of opp. coins)** in the cluster. This is because more the number of our coins, more is the chance that one of our coins gets pocketed and more the number of opp. coins, more is the chance of a foul. Negative priority clusters are ignored.

Force for both cluster and cut-shots are similar to one player case except that constants have been fine-tuned more carefully to avoid fouls (by pocketing opp. coin or striker).

iv) In case neither cut-shot nor any favourable clusters are present, it means that we have our pieces spread apart and obstructed by coins of opp. color. In this case we aim at the nearest coin
attempting to remove the pieces blocking it. Our goal is to change the state of the board so that in the next turn a cut-shot or a cluster-shot is available.

v) Queen capture - Queen is not given any special priority as in one-player case, except when there are only two of our coins left including the queen. In that case, the queen is exclusively targeted until she gets captured. There is no merit to capturing the queen beforehand according to the current rules of the game, hence it is ignored until we have only two of our coins left. If the opponent captures the queen in the meantime, that works to our benefit since we no longer have to do that and and the opponent has wasted some turns in pocketing and covering it.

Other notes:
The major difference between 1-player and 2-player scenarios is the priority associated with cluster-breaking and cut-shots. In 1-player case, cluster breaking was prioritized because a lot of coins could be pocketed in the process when hit with sufficient force and aimed correctly. In 2-player case clusters exclusive to our coins are rarely found, there usually exists coins of opp type in the cluster also which when pocketed results in a foul. In contrast trying to pocket an unobstructed coin using cut-shot is more favourable (because chance of foul is less).

We compared the performance of two 2-player agents vs each other, first one which prioritized cut-shots over cluster-breaking and the second one which prioritized the reverse.

Results (for 50 games) using the script generate-statsP2.sh :

i) First agent is Player 1, second is player 2.
Player 1 won: .47 % games

ii) First agent is Player 2, second is Player 1
Player 1 won: .33 % games

Combining the results of both experiments we see that our prioritizing cut-shots over cluster-shots is indeed better.

: