Feature-Based Reinforcement Learning for the Rubik Cube Puzzle

Created By:

Paromita Banerjee (paro10b@uw.edu)
Anirudh Rao (anirao26@uw.edu)

Guidance:

Prof. Steve Tanimoto

This study is conducted as a part of curriculum assignment for CSE 415 (Introduction to Artificial Intelligence). Any questions and concerns related to this study should be raised to paro10b@uw.edu, anirao26@uw.edu.

<u>Problem Statement:</u>

The program models a Rubik's cube puzzle as a Markov Decision Process and implements feature-based reinforcement learning to solve it. Specifically, we implemented the SARSA algorithm to achieve this.

<u>Technique used:</u>

<u>Algorithm:</u>

SARSA (Current **S**tate, Current **A**ction, **R**eward, Next **S**tate, Next **A**ction)
We used SARSA, which is an on-policy algorithm for Temporal Difference Learning. The major difference between SARSA and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. In SARSA, updates are done using the quintuple Q(s, a, r, s', a'), where: s, a are the original state and action, r is the reward observed in the following state and s', a' are the new state-action pair. One of the primary advantages of using the SARSA algorithm over Q-Learning is that we do not need to know all possible state, action pairs from the beginning. Instead, the agent explores its environment and actions are based on an exploration strategy, which decides the future course of action.

<u>Features Used:</u>

1. Number of completed faces: The count of the faces that are complete, i.e., all values are same.

2. Correct numbers in any face: Each face is identified by a unique value and this feature is an indication of how many quadrants in a face have the unique value of the face assigned to them.

For e.g- If the "Top" Face consists of 1,1,2,1,3,4, then the correct numbers in the Top face are 3, which is the count of 3 1s on that face, since "Top" face is identified by the value "1".

Exploitation Technique:

Epsilon Greedy was selected as the exploitation strategy to choose an action from the current state to transition to a new state. We used the policy learnt by the agent to choose the action with the maximum Q value at the new state.

Final Hyperparameter Values:

- Discount (Gamma): 0.9
- Learning Rate: 0.05
- No. of Episodes: 100
- Epsilon: 0.2

Maximum Moves per Episode:

For each episode, we have restricted the maximum number of moves to 50. This is because depending on the initial state the Agent may take a very long time to reach the goal state

The following are the list of moves that we used:
‘front-face-right', 'back-face-right', 'left-face-right', 'right-face-right', 'top-face-right', 'bottom-face-right', 'front-face-left', 'back-face-left’, 'left-face-left', 'right-face-left', 'top-face-left', 'bottom-face-left

For e.g.- In ‘front-face-right’, “front” signifies the reference face, “right” signifies the direction of rotation. “right” signifies clockwise rotation while “left” signifies anti-clockwise rotation.

State Space Representation:

Each face of the state is represented as follows:

```
FACE1 = "TOP”,
FACE2 = "LEFT”,
FACE3 = "RIGHT”,
FACE4 = "BOTTOM”,
FACE5 = "BACK”,
FACE6 = "FRONT"
```

Cube Size: 2*2*2 =8 cubies

Allowed Moves: All moves – 90 degree, 180 degree, 270 degree, 360 degree

Face representation: Every face is represented by a number

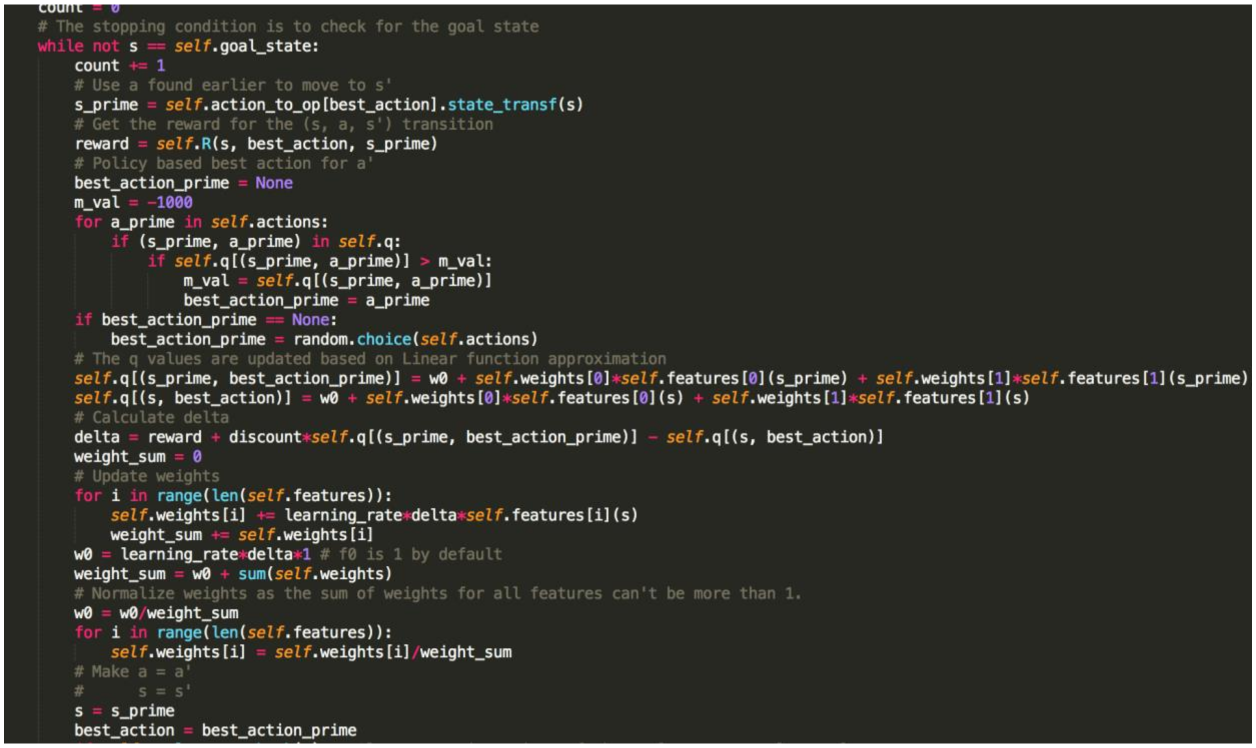
Start State Restriction: Any state can be the start state, since all moves are allowed

Transcript/Screenshot of a session:

Demo Instructions:

To run the code, simply run the file called 'Run.py'. It will display the goal state and ask the user to input the number of moves by which it will scramble the goal state to a start state.

Code Excerpt:



```python
count = 0
# The stopping condition is to check for the goal state
while not s == self.goal_state:
    count += 1
    # Use a found earlier to move to s'
    s_prime = self.action_to_op[best_action].state_transf(s)
    # Get the reward for the (s, a, s') transition
    reward = self.R(s, best_action, s_prime)
    # Policy based best action for a'
    best_action_prime = None
    m_val = -1000
    for a_prime in self.actions:
        if (s_prime, a_prime) in self.q:
            if self.q[(s_prime, a_prime)] > m_val:
                m_val = self.q[(s_prime, a_prime)]
                best_action_prime = a_prime
    if best_action_prime == None:
        best_action_prime = random.choice(self.actions)
    # The q values are updated based on Linear function approximation
    self.q[(s_prime, best_action_prime)] = w0 + self.weights[0]*self.features[0](s_prime) + self.weights[1]*self.features[1](s_prime)
    self.q[(s, best_action)] = w0 + self.weights[0]*self.features[0](s) + self.weights[1]*self.features[1](s)
    # Calculate delta
    delta = reward + discount*self.q[(s_prime, best_action_prime)] - self.q[(s, best_action)]
    weight_sum = 0
    # Update weights
    for i in range(len(self.features)):
        self.weights[i] += learning_rate*delta*self.features[i](s)
        weight_sum += self.weights[i]
    w0 = learning_rate*delta*1 # f0 is 1 by default
    weight_sum = w0 + sum(self.weights)
    # Normalize weights as the sum of weights for all features can't be more than 1.
    w0 = w0/weight_sum
    for i in range(len(self.features)):
        self.weights[i] = self.weights[i]/weight_sum
    # Make a = a'
    #      s = s'
    s = s_prime
    best_action = best_action_prime
```

The above code is an implementation of SARSA. The algorithm looks at the current state and chooses an action based on the epsilon greedy algorithm. It finds the successor state 's_prime' based on the action chosen at state 's'. Once the agent transitions to s_prime, the best action is selected based on the policy followed by the agent. The Q values are updated calculating the weighted sum of the features chosen. Weight normalization ensures the sum of weight is equal to 1. Once this is done, the weights are updated according to learning rate. Finally, the current state and best action are updated with a successor state and future action.

Lessons Learnt:

- Problem Formulation for a Rubik's cube
- SARSA was a new Reinforcement Learning technique that we learnt to solve the cube
- Advantages of SARSA over Q-Learning
- Feature Engineering for a Rubik's cube

Future Steps:

In the future, we would like to:

- Generalize our agent to be able to solve a cube with more than 2 faces.
- Enhance our feature space
- Tweak the SARSA algorithm further to perform better

References:

Name - Artificial Intelligence, Poole & Mackworth (LCI, UBC, Vancouver, Canada)
URL- http://artint.info/html/ArtInt_272.html
The website helped us understand SARSA. We built our Q-Learning function using the algorithm mentioned in the website.

Name - Evolving policies to solve the Rubik's Cube: Experiments with ideal and approximate performance functions
URL-https://dalspace.library.dal.ca/bitstream/handle/10222/72115/Smith-Robert-MCSC-CSCI-August-2016.pdf?sequence=3
The website helped us understand the difference between SARSA & Q-Learning. We also got an idea about representing our states & actions from the website.