CSE 415 Final Project

Option 3: Feature-based reinforcement learning

Shade Wilson (solo)

Usage: The program (Q_Learn.py) attempts to solve a NxN Rubik's cube using Q learnings. It is reliably able to solve a scrambled 2x2 cube (using 180 degree moves) in ~5000 iterations and can occasionally solve a 3x3 cube if said cube is not far from the solved state

The program is based around the data representation of the rubik's cube I built in **cube.py**. Each individual block or cubie is represented as a 3-dimentional vector. This simplifies rotations and allows for easier adaptation of variable sized cubes. 180 degree rotation operators are used for move from one state to another. The main AI technique used was Q learning, implemented in **Q_Learn.py**. The driver launches a set of transitions for perform specified by the user. During each transition, an action is applied to a state to return a new state. This is either a random action or the optimal action according to the policy, depending on epsilon. Epsilon is slowly decreased as the process continues in order to exploit the results of learning. With each transition, the Q values is updated. For feature-based reinforcement learning, this means that the weights for each state feature are updated as well. I had issues with the weights exploding to NaN, so I constrained them to [-1, 1]. The features are important as with the Rubik's cube, the state space is far too large to fully explore, so q values need to be approximated by the features of that state.

Transcript

This is running 1000 iterations with 5 repeats of q learning on a scrambled 2x2. W0-W7 are the values of the weights for the features. Here the 2x2 was solved in 4 moves, which seems like about the average from what I've seen. The toString method is not the best, but it allows some insight into the state of the cube.

```
PARAMETERS:
ALPHA: 0.015625; EPSILON: 0.03125, GAMMA: 0.8
Living reward: -0.1
w0: 0.49997930420663206
w1: 0.9999586084132641
w2: 0.7405141411624823
w3: 0.9999586084132641
w4: 1.0
w5: 0.48360956360747337
w6: 1.0
w7: 0.5200355228909179
Converged!
Path:
Initial state:
Front: ROOR
Back:  ROOR
Up:    YYYY
Down:  WWWW
Left:  BGGB
Right: BGGB

Rotate 180'L (196.5486187762951)
Initial state:
Front: OROR
Back:  OROR
Up:    WWYY
Down:  YYWW
Left:  BGGB
Right: BGGB

Rotate 180'B (256.01953354071645)
Initial state:
Front: OROR
Back:  RORO
Up:    WWYY
Down:  YYWW
Left:  BBGG
Right: BBGG

Rotate 180'R (312.42105363914925)
Initial state:
Front: OROR
Back:  RORO
Up:    WWWW
Down:  YYYY
Left:  BBGG
Right: GGBB

Rotate 180'D (399.87771185210744)
Initial state:
Front: RRRR
Back:  OOOO
Up:    WWWW
Down:  YYYY
Left:  GGGG
Right: BBBB

Exit (499.9777529339498)
```

Demo instructions

Q_Learn.py is the engine of the q learning. Usage:

Usage: python Q_Learn.py [N] [n_transitions] [n_repeats] [level (0 - 3)],

Where N is the size of the cube, n_transitions is the number of transitions to perform for each repeat (n_repeats), and level is which level to select. Level 0 is one step from the goal state, 1 is two steps away, 2 is several, and 3 is scrambled.

Example call: python Q_Learn.py 2 1000 5 3

Code exerpt

The following is an excerpt from Q_Learn.py. It's a function that returns the q value for a state (s), action (a) pair if one has already been observed. If we've never seen the state and/or action, a q value is calculated using the weighted features.

```
205  def compute_q_value(s, a):
206      """Compute Q values, specific to feature-based Q learning.
207
208      If we've visited a state and performed an action before,
209      return the actual Q value. Otherwise, compute the
210      approximate q value based on the features.
211      """
212      global Q_VALUES
213
214      if s in Q_VALUES and a in Q_VALUES[s]:
215          #print("Seen this (s, a) pair: {}".format(Q_VALUES[s][a]))
216          return Q_VALUES[s][a]
217
218      res = 0.0
219      for i in range(len(WEIGHTS)):
220          res += WEIGHTS[i] * FEATURES[i](s, a)
221
222      if not s in Q_VALUES:
223          Q_VALUES[s] = {}
224
225      Q_VALUES[s][a] = res
226
227      return res
```

I learned a lot during this project, but one thing that stands out is that feature selection is very difficult! Summarizing the state of the cube in a way that's informative and pushes the agent to solve the puzzle was not an easy task. Had I had move time, I would've continued working to improve the features for

the 3x3. I believe that the reason my bot can't solve a 3x3 is because the features are specific enough. Perhaps something like a pattern database or some sort of other heuristic would have helped.

Citations

Several sources helped guide me in writing the q learning algorithms:

https://artint.info/html/ArtInt_272.html

https://courses.cs.washington.edu/courses/cse415/19sp/assign/A6.html

https://inst.eecs.berkeley.edu/~cs188/fa10/slides/FA10%20cs188%20lecture%2012%20--%20reinforcement%20learning%20II%20(6PP).pdf

For the representation of the cube, I had some help getting started from here:

https://softwareengineering.stackexchange.com/questions/142760/how-to-represent-a-rubiks-cube-in-a-data-structure

For wrapping my head around the larger ideas of reinforcement learning, I used this article:

https://medium.com/datadriveninvestor/reinforcement-learning-to-solve-rubiks-cube-and-other-complex-problems-106424cf26ff

These pages helped me come up with ideas for features:

https://www.youcandothecube.com/solve-it/2-x-2-solution

https://www.youcandothecube.com/solve-it/3-x-3-solution