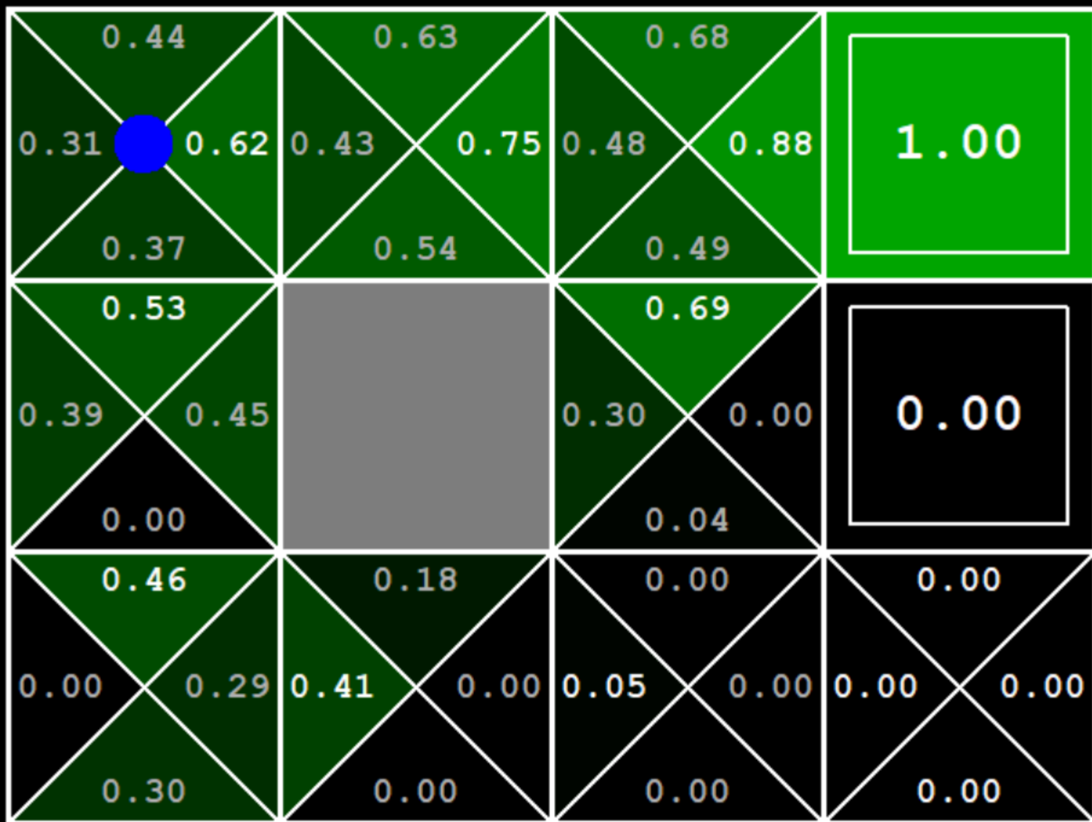
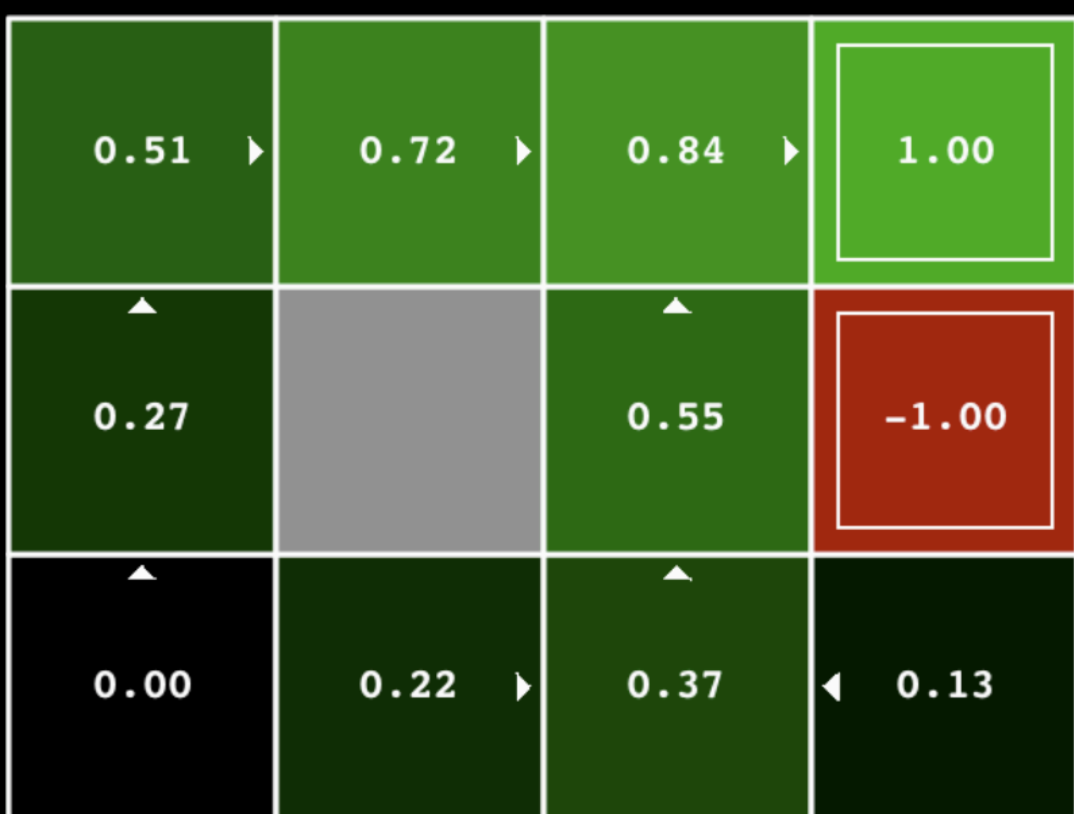
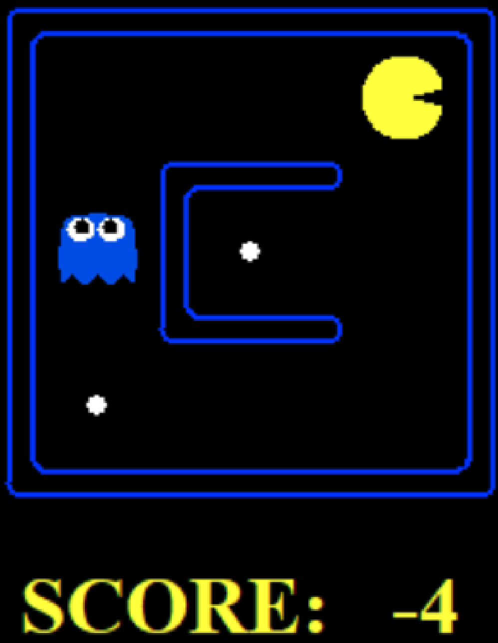


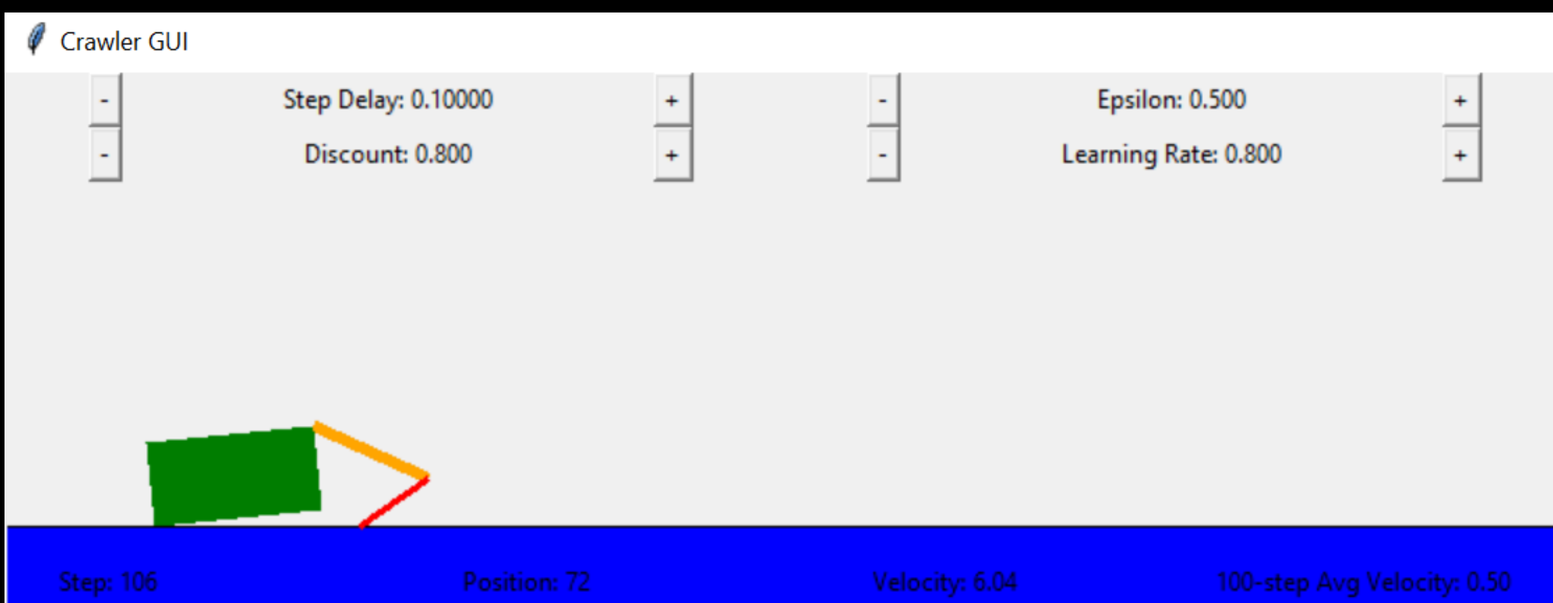
CURRENT Q-VALUES



VALUES AFTER 5 ITERATIONS




```
Reinforcement Learning Status:
        Completed 500 out of 2000 training episodes
        Average Rewards over all training: -425.94
        Average Rewards for last 100 episodes: -319.48
        Episode took 1.28 seconds
Reinforcement Learning Status:
        Completed 600 out of 2000 training episodes
        Average Rewards over all training: -391.82
        Average Rewards for last 100 episodes: -221.21
        Episode took 1.24 seconds
Reinforcement Learning Status:
        Completed 700 out of 2000 training episodes
        Average Rewards over all training: -371.41
        Average Rewards for last 100 episodes: -248.99
        Episode took 1.24 seconds
```



$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_k(s')\right]$$

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

$$\text{sample} = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot \text{sample}$$

```python
class ValueIterationAgent(ValueEstimationAgent):

        A ValueIterationAgent takes a Markov decision process
        (see mdp.py) on initialization and runs value iteration
        for a given number of iterations using the supplied
        discount factor.
    """
    def __init__(self, mdp: mdp.MarkovDecisionProcess, discount = 0.9, iterations = 100):
        """
          Your value iteration agent should take an mdp on
          construction, run the indicated number of iterations
          and then act according to the resulting policy.

          Some useful mdp methods you will use:
              mdp.getStates()
              mdp.getPossibleActions(state)
              mdp.getTransitionStatesAndProbs(state, action)
              mdp.getReward(state, action, nextState)
              mdp.isTerminal(state)
        """
        self.mdp = mdp
        self.discount = discount
        self.iterations = iterations
        self.values = util.Counter() # A Counter is a dict with default 0
        self.runValueIteration()

    def runValueIteration(self):
        """
          Run the value iteration algorithm. Note that in standard
          value iteration, V_k+1(...) depends on V_k(...)'s.
        """
        "*** YOUR CODE HERE ***"
        for i in range(self.iterations):
            tempStateSpace = util.Counter()
            for state in self.mdp.getStates():
                #go over all actions per state
                if not self.mdp.isTerminal(state):
                    actions = self.mdp.getPossibleActions(state)
                    tempStateSpace[state] = max([self.computeQValueFromValues(state, action) for action in actions])
                else:
                    self.values[state] = 0
            self.values = tempStateSpace

    def getValue(self, state):
        """
          Return the value of the state (computed in __init__).
        """
        return self.values[state]

    def computeQValueFromValues(self, state, action):
        """
          Compute the Q-value of action in state from the
          value function stored in self.values.
        """
        "*** YOUR CODE HERE ***"
        QValue = 0
        list = self.mdp.getTransitionStatesAndProbs(state,action)
        for stateProbPair in list:
            sPrime = stateProbPair[0]
            prob = stateProbPair[1]
            QValue += prob * (self.mdp.getReward(state, action, sPrime) + (self.discount * self.values[sPrime] \
                if sPrime in self.values and not self.mdp.isTerminal(sPrime) else 0))

        return QValue
```

```python
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    bestAction = None
    bestQValue = float('-inf')

    if self.mdp.isTerminal(state):
        return bestAction

    for action in self.mdp.getPossibleActions(state):
        newQValue = self.computeQValueFromValues(state, action)

        if newQValue > bestQValue:
            bestQValue = newQValue
            bestAction = action

    return bestAction
```

```python
class ApproximateQAgent(PacmanQAgent):
    """
       ApproximateQLearningAgent
       You should only have to overwrite getQValue
       and update.  All other QLearningAgent functions
       should work as is.
    """
    def __init__(self, extractor='IdentityExtractor', **args):
        self.featExtractor = util.lookup(extractor, globals())()
        PacmanQAgent.__init__(self, **args)
        self.weights = util.Counter()

    def getWeights(self):
        return self.weights

    def getQValue(self, state, action):
        """
          Should return Q(state,action) = w * featureVector
          where * is the dotProduct operator
        """
        "*** YOUR CODE HERE ***"
        features = self.featExtractor.getFeatures(state,action)

        featuresList = sum([self.weights[index] * weight for index, weight in features.items()])
        return featuresList


    def update(self, state, action, nextState, reward: float):
        """
           Should update your weights based on transition
        """
        "*** YOUR CODE HERE ***"
        features = self.featExtractor.getFeatures(state,action)
        diff = (reward + self.discount * self.computeValueFromQValues(nextState)) - self.getQValue(state,action)

        for feature in features.keys():
            print(feature)
            self.weights[feature] += self.alpha * diff * features[feature]
```