ESIEE Paris

Artificial Intelligence and Cybersecurity

Artificial Intelligence

Lab 4 Report

Antoine Debauge
Bruno Luiz Dias Alves de Castro
Victor Gabriel Mendes Sündermann

April 23, 2023

Contents

1 Introduction **2**

2 Value Iteration **2**

3 Q-Learning **4**

4 Approximate Q-Learning **8**

5 Conclusion **9**

1 Introduction

In this report we will present the results of the fourth project of the Artificial Intelligence course. The project consists of implementing reinforcement learning algorithms and agents to score actions on a grid and eventually solve the **Pacman** game. The topics worked on this lab were: Value Iteration, Q-Learning, and aproximate Q-Learning.

2 Value Iteration

The code implemented for this algorithm can be seen in Tables 1, 2, and 3.

```
def runValueIteration(self):
  """
    Run the value iteration algorithm. Note that in standard
    value iteration, V_k+1(...) depends on V_k(...)'s.
  """

  for i in range(self.iterations):
      values = self.values.copy()
      for state in self.mdp.getStates():
          if self.mdp.isTerminal(state):
              continue

          max_value = float("-inf")
          for action in self.mdp.getPossibleActions(state):
              max_value = \
                max(max_value, self.computeQValueFromValues(state, action))
          values[state] = max_value
      self.values = values
```

Table 1: runValueIteration function

```
def computeQValueFromValues(self, state, action):
  """
    Compute the Q-value of action in state from the
    value function stored in self.values.
  """

  q_value = 0
  for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action):
      q_value += prob * (self.mdp.getReward(state, action, next_state) \
      + self.discount * self.values[next_state])

  return q_value
```

Table 2: computeQValueFromValues function

To run the program, we use the following command:

```
python3 gridworld.py -a value -i 5
```

This test created the expected output as shown in the following image:
Then we tested the code in a bigger scale and executing the policies more times, with the command:

```
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.
  """

  if self.mdp.isTerminal(state):
      return None

  max_value = float("-inf")
  best_action = None
  for action in self.mdp.getPossibleActions(state):
      q_value = self.computeQValueFromValues(state, action)
      if q_value > max_value:
          max_value = q_value
          best_action = action

  return best_action
```

Table 3: computeActionFromValues function

Table 4: GridWorld with Value Iteration

```
python3 gridworld.py -a value -i 100 -k 10
```

The result can be seen bellow:

Table 5: GridWorld with Value Iteration k=100

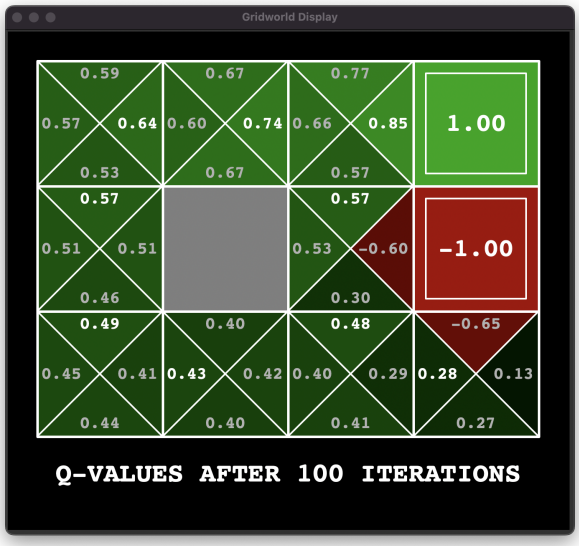

Table 6: Q-Values after 100 iterations

3 Q-Learning

The objective of this section is implementing a Q-Learning agent that learns through trial and error, this agent is constitued of five functions: init(), update(), computeValuefromQValues(), getQValue() and computeActionFromQValues(). The implementation of each of those functions can be seen bellow:

```
def __init__(self, **args):
"You can initialize Q-values here..."
ReinforcementAgent.__init__(self, **args)

# Q-values
self.qValuesDict = {}
```

Table 7: init function

```
def update(self, state, action, nextState, reward: float):
"""
  The parent class calls this to observe a
  state = action => nextState and reward transition.
  You should do your Q-Value update here
  NOTE: You should never call this function,
  it will be called on your behalf
"""

# Compute sample
sample = reward + self.discount * self.computeValueFromQValues(nextState)

# Update Q-value
self.qValuesDict[(state, action)] = (1 - self.alpha) * \
  self.getQValue(state, action) + self.alpha * sample
```

Table 8: update function

```
def computeValueFromQValues(self, state):
"""
  Returns max_action Q(state,action)
  where the max is over legal actions.  Note that if
  there are no legal actions, which is the case at the
  terminal state, you should return a value of 0.0.
"""

legalActions = self.getLegalActions(state)
if len(legalActions) == 0:
    return 0.0

maxQValue = -math.inf
for action in legalActions:
    qValue = self.getQValue(state, action)
    if qValue > maxQValue:
        maxQValue = qValue

return maxQValue
```

Table 9: computeValuefromQValues function

```
def getQValue(self, state, action):
"""
  Returns Q(state,action)
  Should return 0.0 if we have never seen a state
  or the Q node value otherwise
"""
if (state, action) not in self.qValuesDict:
    self.qValuesDict[(state, action)] = 0.0

return self.qValuesDict[(state, action)]
```

Table 10: getQValue function

Now that the Q-Learning agent is done it's time to test it with the following command:

```
def computeActionFromQValues(self, state):
"""
  Compute the best action to take in a state.  Note that if there
  are no legal actions, which is the case at the terminal state,
  you should return None.
"""

legalActions = self.getLegalActions(state)
if len(legalActions) == 0:
    return None

maxQValue = -math.inf
bestAction = None
for action in legalActions:
    qValue = self.getQValue(state, action)
    if qValue > maxQValue:
        maxQValue = qValue
        bestAction = action

return bestAction
```

Table 11: computeActionFromQValues function

```
python3 gridworld.py -a q -k 5 -m
```

We controlled the agent manually throught the grid and achieved the following result:

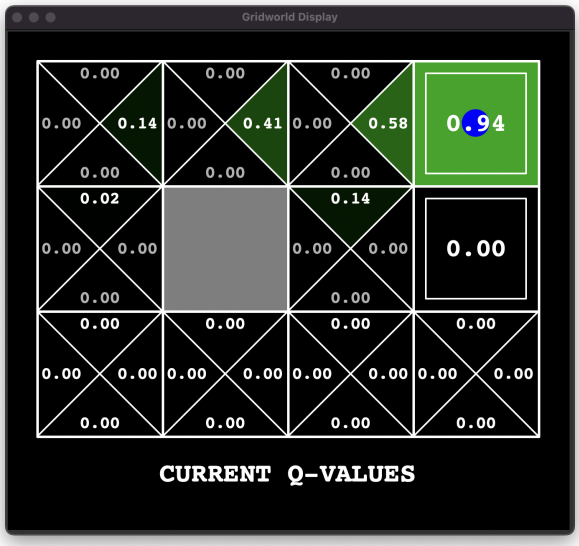

Table 12: Q-Learning agent with manual iteration

Now we'll add the episilon-greedy cut to automate our agent and see how it performs, for this we ran the following command, which executes the agent 100 times:

```
python3 gridworld.py -a q -k 100
```

After the simulation we got the following result:

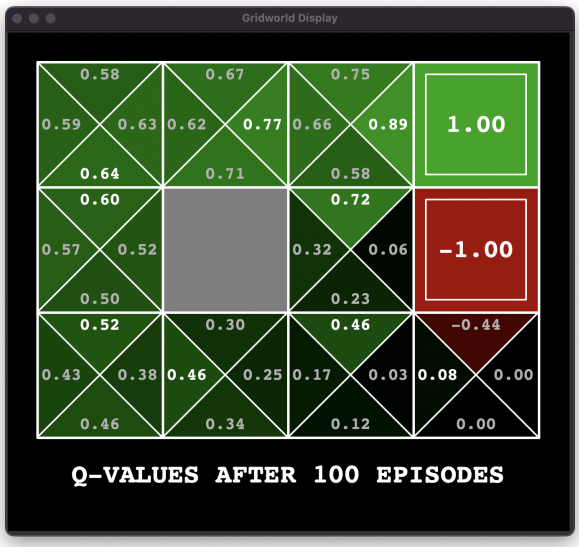

Table 13: Q-Learning agent with episilon-greedy cut

We can see that the agent sucessfully achieved the goal multiple times and changed the score of various squares of the grid. After seeing that it's functional we ran the crawler.py simulation and saw our agent evolving and reaching the end of the map, as seen bellow:

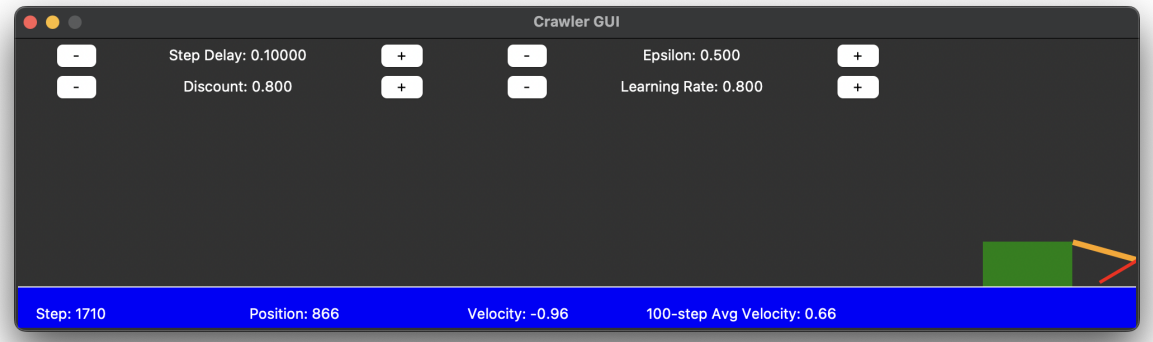

Table 14: Q-Learning agent in crawler.py simulation

The final step for the Q-Learning agent was to apply it to the Pacman game, for this we ran the following command:

```
python3 pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

The first 2000 interations are used to train the agent and the last 10 are used to test it, the table bellow shows the reults of the training:

After the training was done the Pacman proceded to testing generating the following results:
As seen in the table the Pacman achieved 100% of the wins, which means that the agent was able to learn the game and play it perfectly.

Completed	Average Rewards over all training	Average Rewards for last 100 episodes	Time per episode
100/2000	-511.96	-511.96	0.20s
200/2000	-512.18	-512.41	0.25s
300/2000	-488.55	-441.29	0.25s
400/2000	-452.35	-343.74	0.31s
500/2000	-423.72	-309.20	0.31s
600/2000	-391.24	-228.81	0.3s
700/2000	-378.26	-300.38	0.32s
800/2000	-366.19	-281.69	0.33s
900/2000	-341.96	-148.20	0.31s
1000/2000	-305.32	24.51	0.31s
1100/2000	-271.73	64.11	0.32s
1200/2000	-240.34	104.97	0.32s
1300/2000	-202.12	256.52	0.31s
1400/2000	-166.45	297.33	0.30s
1500/2000	-140.89	216.90	0.30s
1600/2000	-115.38	267.19	0.31s
1700/2000	-91.70	287.31	0.31s
1800/2000	-71.26	276.20	0.32s
1900/2000	-55.58	226.59	0.31s
2000/2000	-39.95	257.10	0.31s

Table 15: Training set Pacman

Result	Scourse
Win	502
Win	499
Win	503
Win	503
Win	495
Win	503
Win	499
Win	503
Win	503
Win	502

Table 16: Test set Pacman

4 Approximate Q-Learning

The last part of this report consists of an implementation of the Approximate Q-Learning algorithm, which is a variation of the Q-Learning algorithm that uses a function to approximate the Q-Values instead of a table. The function used in this implementation is a linear combination of the features of the state, which are the position of the Pacman and the position of the ghosts. The features are represented by a vector of 5 elements, the first 2 elements are the position of the Pacman and the last 3 elements are the position of the ghosts. The weights of the features are represented by a vector of 5 elements, the first 2 elements are the weights of the Pacman position and the last 3 elements are the weights of the ghosts position. The Q-Value of a state is the dot product of the features vector and the weights vector. The implementation of the Approximate Q-Learning algorithm is shown in the code bellow:

To test our agent we ran the following command, which simulates 10 games their results are shown in the table bellow:

```
python3 pacman.py -p ApproximateQAgent -a \
extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid
```

```
def getQValue(self, state, action):
"""
  Should return Q(state,action) = w * featureVector
  where * is the dotProduct operator
"""
Q = 0

for i in self.featExtractor.getFeatures(state, action):
    Q += self.weights[i] * self.featExtractor.getFeatures(state, action)[i]

return Q
```

Table 17: getQValue function

```
def update(self, state, action, nextState, reward: float):
"""
   Should update your weights based on transition
"""
difference = (reward + self.discount * \
  self.computeValueFromQValues(nextState)) - self.getQValue(state, action)
for i in self.featExtractor.getFeatures(state, action):
    self.weights[i] += self.alpha * difference \
      * self.featExtractor.getFeatures(state, action)[i]

return
```

Table 18: update function

```
def final(self, state):
"""Called at the end of each game."""
# call the super-class final method
PacmanQAgent.final(self, state)

# did we finish training?
if self.episodesSoFar == self.numTraining:
    # you might want to print your weights here for debugging
    print(self.weights)
    pass
```

Table 19: final function

5 Conclusion

By programming and implementing our own methods and agents, we were able to fully understand the functionings of the reinforcement learning and how it is an essential aspect in the domain of Artificial Inteligence. On top of that we were able to understand the importance of the Markov Decision Process and how it is used in the reinforcement learning.

Result	Scourse
Win	527
Win	529
Win	525
Win	525
Win	527
Win	529
Win	527
Win	529
Win	527
Win	527

Table 20: Test set Pacman