[CSE537] Reinforcement Learning

REPORT for TEAM PROJECT #5

by Oleksii Starov, Hyungjoon Koo

Dec. 6, 2013

A. General Information

(1) Team-up (Group 41): Oleksii Starov (ostarov@cs.stonybrook.edu), Hyungjoon Koo (hykoo@cs.stonybrook.edu)

(2) Project information: Classification (PJT#5), http://www.cs.sunysb.edu/~ram/cse537/project05-2013.html (Due date: Dec 6, 2013)

(3) Implementation: MDPs, Q-Learning (Extra)

B. Implementation Note

(1) *ValueIterationAgent* Class (Mandatory part)

The class of *ValueIterationAgent* has five methods: *__init__(mdp, discount, iterations), getValues(state), getQValue(state, action), getPolicy(state)* and *getAction(state)*. We first declare a temporary value dictionary to store the values for the next iteration and then compute the Bellman Equation per each action and state. The iterated values are updated to the previous values so that it allows the next iteration to use those values. The value iteration is totally based upon the general form of the equation as following.

$$Q^*(\mathrm{s},\mathrm{a}) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

$$V^*(\mathrm{s}) = max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

```
class ValueIterationAgent(ValueEstimationAgent):

  def __init__(self, mdp, discount = 0.9, iterations = 100):

    self.mdp = mdp
    self.discount = discount
    self.iterations = iterations
    self.values = util.Counter() # A Counter is a dict with default 0

    "*** YOUR CODE HERE ***"

    allStates =  mdp.getStates()

    for i in range(0, iterations) :

      # Declare a temporary value dictionary to store the values for the next iteration
      iterationValues = self.values.copy()
      for eachState in allStates :
        maxValue = -99999
        # Compute [V(s) = max Q value] for each (action, state) per the iteration
        for eachAction in mdp.getPossibleActions(eachState) :
          currentQValue = self.getQValue(eachState, eachAction)
          if (currentQValue >= maxValue) :
            maxValue = currentQValue
```

```
                iterationValues[eachState] = currentQValue

          # Copy updated values to the original value list
          #   so that the next iteration can make a reference (Dynamic Programming)
          self.values = iterationValues.copy()
          print self.values

  def getValue(self, state):
    return self.values[state]


  def getQValue(self, state, action):

    "*** YOUR CODE HERE ***"

    # Q Value is the expected utility having taken action "a" from state "s" to "s'"
    # Q value = [sum of (T(s,a,s')*{R(s,a,s')+(r*V(s'))}] where
    # T(s,a,s'): the probability for the next state with a certain action = mdp.getTransitionStatesAndProbs(state, action)
    # R(s,a,s'): the reward for the next state with a certain action = mdp.getReward(state, action, nextState)
    # r or gamma: self.discount, V(s'): v value of next state

    qValue = 0.0
    currentTransitionInfo = self.mdp.getTransitionStatesAndProbs(state, action)
    for (nextState, nextStateProb) in currentTransitionInfo :
       qValue +=  nextStateProb * (self.mdp.getReward(state, action, nextState) + self.discount * self.getValue(nextState))

    # Returns the q-value of the (state, action) pair
    return qValue

  def getPolicy(self, state):

    "*** YOUR CODE HERE ***"

    # If the given state is a terminal one, then return None
    if self.mdp.isTerminal(state) :
        return None

    # Get the all possible actions with qValue
    # And then return the action which represents the maximum argument
    else :
        actionList = util.Counter()
        actions = self.mdp.getPossibleActions(state)
        for action in actions:
            actionList[action] = self.getQValue(state, action)
        return actionList.argMax()

  def getAction(self, state):
    return self.getPolicy(state)
```

(2) **Q-Learning – *QLearningAgent* Class (Optional Part)**

The class of *ValueIterationAgent* has six methods: *__init__(**args), getValue(state), getQValue(state, action), getPolicy(state), getAction(state)* and *update(state, action, nextState, reward)*. As the problem stated, the q-learning agent is supposed to learn by trial and error from interactions with the environment. The following are the main features: (1) For *getValue* and *getPolicy*, we break ties randomly for better behavior. (2) It can only access Q values by calling *getQValue* from *getValue* and *getPolicy* functions. (3) It allows the agent to epsilon-greedy action selection in *getAction*. (4) It works for crawler.py, which means that it fits into the general MDP cases.

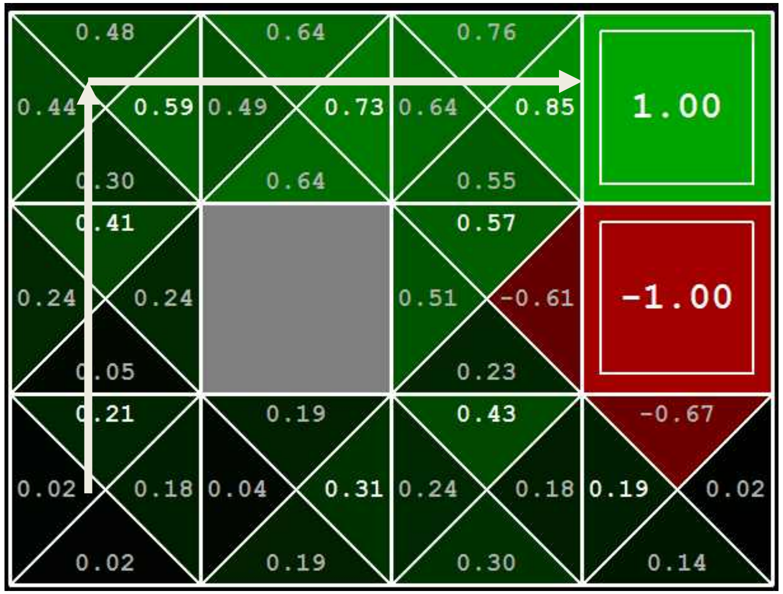
```
class QLearningAgent(ReinforcementAgent):
  """
   Q-Learning Agent

   Functions you should fill in:
     - getQValue
     - getAction
     - getValue
     - getPolicy
     - update

   Instance variables you have access to
     - self.epsilon (exploration prob)
     - self.alpha (learning rate)
     - self.discount (discount rate)

   Functions you should use
     - self.getLegalActions(state)
       which returns legal actions
       for a state
  """

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

    "*** YOUR CODE HERE ***"
    self.qValues = util.Counter()
    self.Policies = {}
    self.actions = {}

  def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we never seen
      a state or (state,action) tuple
    """
    "*** YOUR CODE HERE ***"
    return self.qValues[(state, action)]

  def getValue(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.
    """
    "*** YOUR CODE HERE ***"
    maxNext = 0
    if state in self.actions.keys():
      for a in self.actions[state]:
        if self.qValues[(state, a)] > maxNext:
          maxNext = self.qValues[(state, a)]

    return maxNext

  def getPolicy(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.
    """

    "*** YOUR CODE HERE ***"
    legalActions = self.getLegalActions(state)
    if len(legalActions) == 0:
      return None

    if state not in self.Policies.keys():
      return random.choice(legalActions)

    return self.Policies[state]

  def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.

      HINT: You might want to use util.flipCoin(prob)
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None

    "*** YOUR CODE HERE ***"
    if len(legalActions) == 0:
      return None

    if util.flipCoin(self.epsilon):
      action = random.choice(legalActions)
```

```
    else:
      action = self.getPolicy(state)

    return action

  def update(self, state, action, nextState, reward):
    """
      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
    """
    "*** YOUR CODE HERE ***"

    # Calculations
    newQvalue = (1-self.alpha) * self.qValues[(state, action)]
    newQvalue += self.alpha * (reward + self.discount * self.getValue(nextState))

    # Save updated Q value for state-action pair
    self.qValues[(state, action)] = newQvalue

    # Save new state-action relation
    if state in self.actions.keys():
      self.actions[state].append(action)
    else:
      self.actions[state] = [action]

    # Update policy
    actionList = util.Counter()
    tmpActions = self.actions[state]
    for a in tmpActions:
      actionList[a] = self.getQValue(state, a)
    actionMaxList = []
    maxValue = actionList[actionList.argMax()]
    for a in actionList:
      if actionList[a] == maxValue:
        actionMaxList.append(a)

    self.Policies[state] = random.choice(actionMaxList)
```

C. The results and the description on *analysis.py*

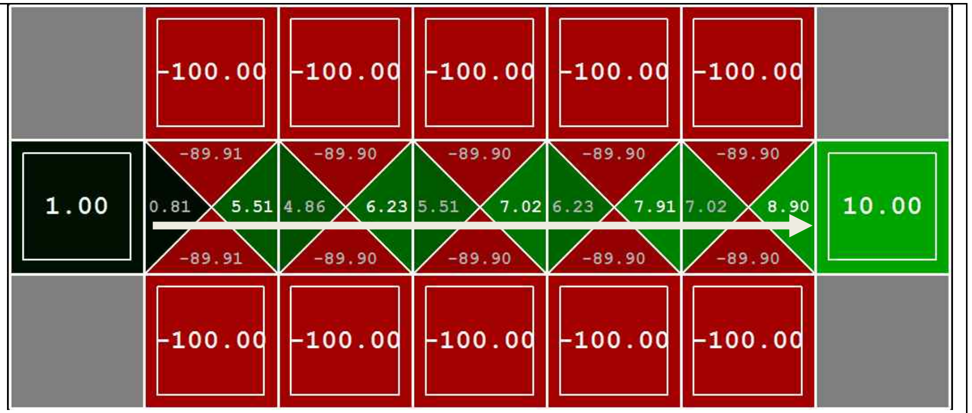
Here are the outcomes and descriptions for each question.

Values along with Value Iteration for Q1	**Q-Values after 5 Iterations for Q1**
=================> Iteration [1] <================= {(0, 1): 0.0, (1, 2): 0.0, (3, 2): 1.0, (0, 0): 0.0, (3, 0): 0.0, (3, 1): -1.0, (2, 1): 0.0, (2, 0): 0.0, (2, 2): 0.0, (1, 0): 0.0, (0, 2): 0.0} =================> Iteration [2] <================= {(0, 1): 0.0, (1, 2): 0.0, (3, 2): 1.0, (0, 0): 0.0, (3, 0): 0.0, (3, 1): -1.0, (2, 1): 0.0, (2, 0): 0.0, (2, 2): 0.7200000000000001, (1, 0): 0.0, (0, 2): 0.0} =================> Iteration [3] <================= {(0, 1): 0.0, (1, 2): 0.5184000000000001, (3, 2): 1.0, (0, 0): 0.0, (3, 0): 0.0, (3, 1): -1.0, (2, 1): 0.42840000000000006, (2, 0): 0.0, (2, 2): 0.7848, (1, 0): 0.0, (0, 2): 0.0} =================> Iteration [4] <================= {(0, 1): 0.0, (1, 2): 0.6583680000000002, (3, 2): 1.0, (0, 0): 0.0, (3, 0): 0.0, (3, 1): -1.0, (2, 1): 0.5136120000000001, (2, 0): 0.30844800000000006, (2, 2): 0.8291880000000001, (1, 0): 0.0, (0, 2): 0.3732480000000001} **=================> Iteration [5] <=================** **{(0, 1): 0.26873856000000007, (1, 2): 0.7155216000000002, (3, 2): 1.0, (0, 0): 0.0, (3, 0): 0.13208256000000002, (3, 1): -1.0, (2, 1): 0.5532404400000002, (2, 0): 0.36980064000000007, (2, 2): 0.8408520000000002, (1, 0): 0.22208256000000004, (0, 2): 0.5076172800000002}**	
Parameter values we found for Q2: answerDiscount = 0.9, answerNoise = 0.001	**Q-Values after 100 Iterations for Q2**
=================> Iteration [1] <================= {(0, 1): 1.0, (3, 2): -100.0, (3, 0): -100.0, (2, 1): 0.0, (5, 1): 0.0, (4, 2): -100.0, (1, 0): -100.0, (4, 0): -100.0, (1, 2): -100.0, (5, 2): -100.0, (6, 1): 10.0, (3, 1): 0.0, (2, 0): -100.0, (5, 0): -100.0, (2, 2): -100.0, (4, 1): 0.0, (1, 1): 0.0} =================> Iteration [2] <================= {(0, 1): 1.0, (3, 2): -100.0, (3, 0): -100.0, (2, 1): -0.09, (5, 1): 8.901, (4, 2): -100.0, (1, 0): -100.0, (4, 0): -100.0, (1, 2): -100.0, (5, 2): -100.0, (6, 1): 10.0, (3, 1): -0.09, (2, 0): -100.0, (5, 0): -100.0, (2, 2): -100.0, (4, 1): -0.09, (1, 1): 0.8090999999999999} ... =================> Iteration [99] <=================	The discount parameter is remained the same (0.9) and the noise parameter has been changed to 0.001. Then the agent is highly likely to move to the direction which the biggest value holds, which ended up with cross the bridge.

{(0, 1): 1.0, (3, 2): -100.0, (3, 0): -100.0, (2, 1): 6.2257087000981715, (5, 1): 8.901, (4, 2): -100.0, (1, 0): -100.0, (4, 0): -100.0, (1, 2): -100.0, (5, 2): -100.0, (6, 1): 10.0, (3, 1): 7.02447858981, (2, 0): -100.0, (5, 0): -100.0, (2, 2): -100.0, (4, 1): 7.912889099999999, (1, 1): 5.507534692258266}

=================> Iteration [100] <=================

{(0, 1): 1.0, (3, 2): -100.0, (3, 0): -100.0, (2, 1): 6.2257087000981715, (5, 1): 8.901, (4, 2): -100.0, (1, 0): -100.0, (4, 0): -100.0, (1, 2): -100.0, (5, 2): -100.0, (6, 1): 10.0, (3, 1): 7.02447858981, (2, 0): -100.0, (5, 0): -100.0, (2, 2): -100.0, (4, 1): 7.912889099999999, (1, 1): 5.507534692258266}



Parameter values we found for Q3(a):
answerDiscount = 0.1, answerNoise = 0.01, answerLivingReward = 0.01

Q-Values after 100 Iterations for Q3(a)
[Prefer the close exit (+1), risking the cliff (-10)]

=================> Iteration [1] <=================

{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.01, (2, 1): 0.01, (2, 3): 0.01, (1, 4): 0.01, (4, 2): 10.0, (0, 3): 0.01, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.01, (3, 3): 0.01, (4, 1): 0.01, (3, 1): 0.01, (4, 4): 0.01, (2, 4): 0.01, (2, 0): -10.0, (4, 3): 0.01, (0, 4): 0.01, (3, 4): 0.01, (0, 2): 0.01}

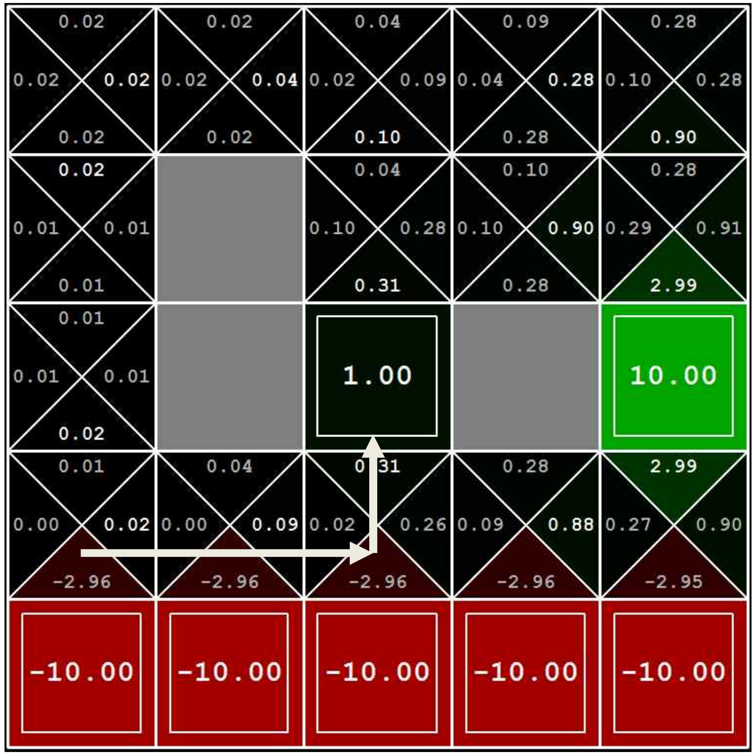
=================> Iteration [2] <=================

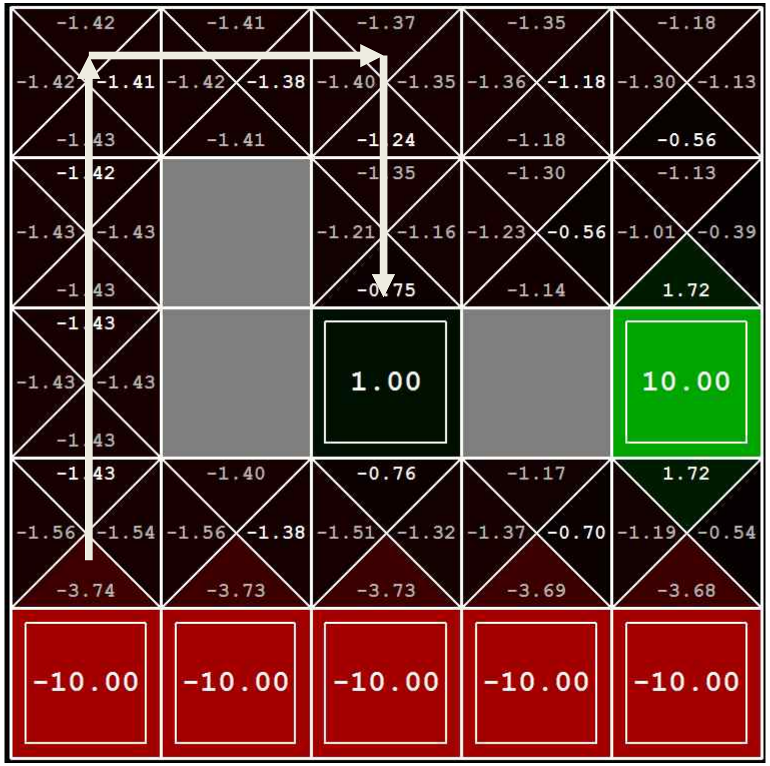
{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.013000000000000003, (2, 1): 0.30703, (2, 3): 0.30703, (1, 4): 0.013000000000000003, (4, 2): 10.0, (0, 3): 0.013000000000000003, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.013000000000000003, (3, 3): 0.013000000000000003, (4, 1): 2.98003, (3, 1): 0.013000000000000003, (4, 4): 0.013000000000000003, (2, 4): 0.013000000000000003, (2, 0): -10.0, (4, 3): 2.98003, (0, 4): 0.013000000000000003, (3, 4): 0.013000000000000003, (0, 2): 0.013000000000000001}

...

=================> Iteration [99] <=================

{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.08674117969605352, (2, 1): 0.3084547741633315, (2, 3): 0.30881105100024314, (1, 4): 0.04047349822190286, (4, 2): 10.0, (0, 3): 0.016607182895341712, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.020786463739432926, (3, 3): 0.8985562824951887, (4, 1): 2.9858033674449547, (3, 1): 0.883108262524939, (4, 4): 0.8985562824951887, (2, 4): 0.1021955479031554, (2, 0): -10.0, (4, 3):



2.9858265742851704, (0, 4): 0.022078657732847437, (3, 4): 0.2786370058335642, (0, 2): 0.016222246470021646} **================> Iteration [100] <================** **{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.08674117969605352, (2, 1): 0.3084547741633315, (2, 3): 0.30881105100024314, (1, 4): 0.04047349822190286, (4, 2): 10.0, (0, 3): 0.016607182895341712, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.020786463739432926, (3, 3): 0.8985562824951887, (4, 1): 2.9858033674449547, (3, 1): 0.883108262524939, (4, 4): 0.8985562824951887, (2, 4): 0.1021955479031554, (2, 0): -10.0, (4, 3): 2.9858265742851704, (0, 4): 0.022078657732847437, (3, 4): 0.2786370058335642, (0, 2): 0.016222246470021646}**	We entered a small discount so that the agent is able to choose the exit (+1).
Parameter values we found for Q3(b): answerDiscount = 0.3, answerNoise = 0.1, answerLivingReward = -1	**Q-Values after 100 Iterations for Q3(b) [Prefer the close exit (+1), but avoiding the cliff (-10)]**
================> Iteration [1] <================ {(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): -1.0, (2, 1): -1.0, (2, 3): -1.0, (1, 4): -1.0, (4, 2): 10.0, (0, 3): -1.0, (1, 0): -10.0, (4, 0): -10.0, (0, 1): -1.0, (3, 3): -1.0, (4, 1): -1.0, (3, 1): -1.0, (4, 4): -1.0, (2, 4): -1.0, (2, 0): -10.0, (4, 3): -1.0, (0, 4): -1.0, (3, 4): -1.0, (0, 2): -1.0} ================> Iteration [2] <================ {(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): -1.3, (2, 1): -0.76, (2, 3): -0.76, (1, 4): -1.3, (4, 2): 10.0, (0, 3): -1.3, (1, 0): -10.0, (4, 0): -10.0, (0, 1): -1.3, (3, 3): -1.3, (4, 1): 1.6700000000000002, (3, 1): -1.3, (4, 4): -1.3, (2, 4): -1.3, (2, 0): -10.0, (4, 3): 1.6700000000000002, (0, 4): -1.3, (3, 4): -1.3, (0, 2): -1.3000000000000003} ... ================> Iteration [99] <================ {(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): -1.3761396774834829, (2, 1): -0.7611021567452988, (2, 3): -0.7496816526935111, (1, 4): -1.3762850424689415, (4, 2): 10.0, (0, 3): -1.4245650405675698, (1, 0): -10.0, (4, 0): -10.0, (0, 1): -1.4274672916057987, (3, 3): -0.5624285268738954, (4, 1): 1.7152689730121355, (3, 1): -0.69733743886977, (4, 4): -0.5624285268738954, (2, 4): -1.2407277451661969, (2, 0): -10.0, (4, 3): 1.717323423448621, (0, 4): -1.4141781087057135, (3, 4): -1.1779615534609749, (0, 2): -1.427456248405406} **================> Iteration [100] <================** **{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): -1.3761396774834829, (2, 1): -0.7611021567452988, (2, 3):**	

-0.7496816526935111, (1, 4): -1.3762850424689415, (4, 2): 10.0, (0, 3): -1.4245650405675698, (1, 0): -10.0, (4, 0): -10.0, (0, 1): -1.4274672916057987, (3, 3): -0.5624285268738954, (4, 1): 1.7152689730121355, (3, 1): -0.69733743886977, (4, 4): -0.5624285268738954, (2, 4): -1.2407277451661969, (2, 0): -10.0, (4, 3): 1.717323423448621, (0, 4): -1.4141781087057135, (3, 4): -1.1779615534609749, (0, 2): -1.427456248405406}

We entered a small discount so that the agent is able to choose the exit (+1) and negative LivingReward so that the agent avoids the cliff. Without negative figure and noise, it tends to take a different route.

Parameter values we found for Q3(c):
answerDiscount = 0.9, answerNoise = 0.01, answerLivingReward = 0.01

Q-Values after 100 Iterations for Q3(c)
[Prefer the close exit (+10), risking the cliff (-10)]

================> Iteration [1] <================

{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.01, (2, 1): 0.01, (2, 3): 0.01, (1, 4): 0.01, (4, 2): 10.0, (0, 3): 0.01, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.01, (3, 3): 0.01, (4, 1): 0.01, (3, 1): 0.01, (4, 4): 0.01, (2, 4): 0.01, (2, 0): -10.0, (4, 3): 0.01, (0, 4): 0.01, (3, 4): 0.01, (0, 2): 0.01}

================> Iteration [2] <================

{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.019000000000000006, (2, 1): 0.9010900000000001, (2, 3): 0.9010900000000001, (1, 4): 0.019000000000000006, (4, 2): 10.0, (0, 3): 0.019000000000000006, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.019000000000000006, (3, 3): 0.019000000000000006, (4, 1): 8.92009, (3, 1): 0.019000000000000006, (4, 4): 0.019000000000000006, (2, 4): 0.019000000000000006, (2, 0): -10.0, (4, 3): 8.92009, (0, 4): 0.019000000000000006, (3, 4): 0.019000000000000006, (0, 2): 0.019000000000000003}

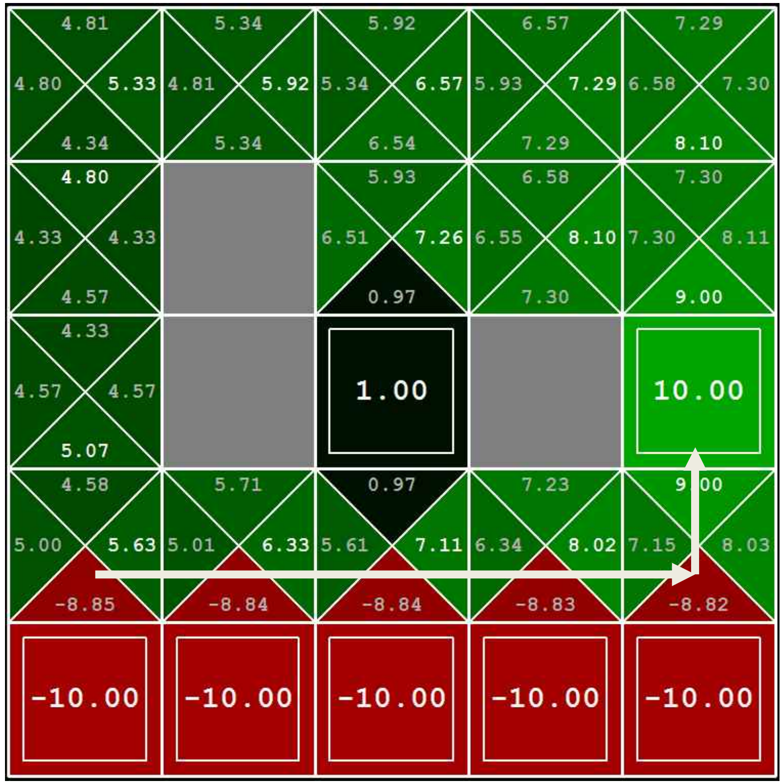
...

================> Iteration [99] <================

{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 6.33086728690785, (2, 1): 7.1126581190985005, (2, 3): 7.257151669710488, (1, 4): 5.916845182561398, (4, 2): 10.0, (0, 3): 4.799986942271567, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 5.628621073719738, (3, 3): 8.095497268004163, (4, 1): 8.99656108102166, (3, 1): 8.017012479347363, (4, 4): 8.095497268004163, (2, 4): 6.569689759728784, (2, 0): -10.0, (4, 3): 8.996915859071843, (0, 4): 5.327482670921574, (3, 4): 7.29233330336286, (0, 2): 5.070738018853972}

================> Iteration [100] <================

{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 6.33086728690785, (2, 1): 7.1126581190985005, (2, 3): 7.257151669710488, (1, 4): 5.916845182561398, (4, 2): 10.0, (0, 3): 4.799986942271567, (1, 0): -10.0, (4, 0): -



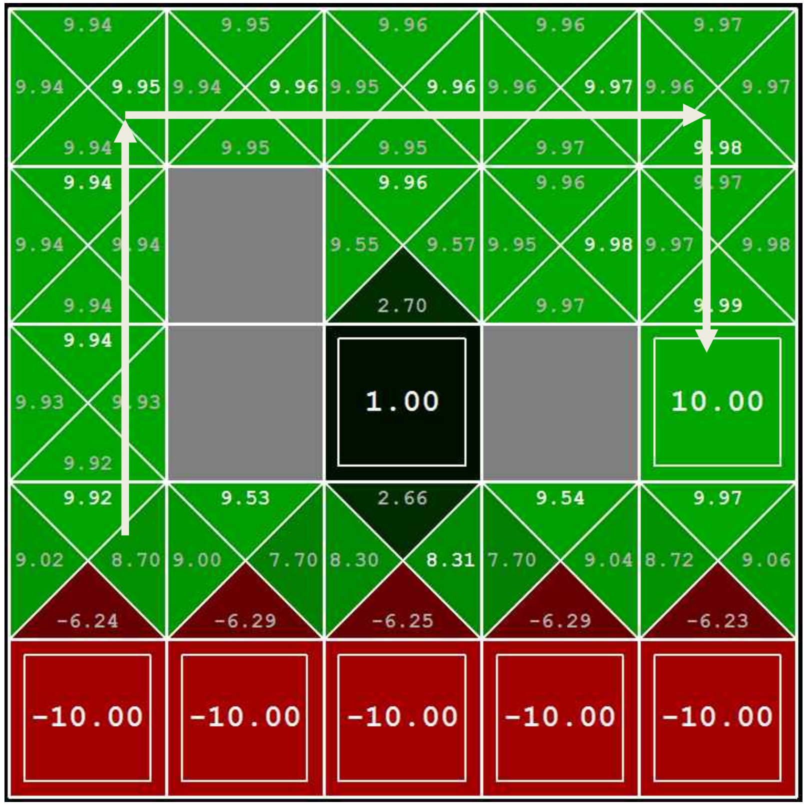
This is a similar case to the 3a(), but we allow the agent to have large discount value in order to choose the exit (+10)

10.0, (0, 1): 5.628621073719738, (3, 3): 8.095497268004163, (4, 1): 8.99656108102166, (3, 1): 8.017012479347363, (4, 4): 8.095497268004163, (2, 4): 6.569689759728784, (2, 0): -10.0, (4, 3): 8.996915859071843, (0, 4): 5.327482670921574, (3, 4): 7.29233330336286, (0, 2): 5.070738018853972}

Parameter values we found for Q3(d): **answerDiscount = 0.9, answerNoise = 0.1, answerLivingReward = 0.99**	**Q-Values after 100 Iterations for Q3(d)** **[Prefer the distant exit (+10), avoiding the cliff (-10)**

=================> Iteration [1] <=================
{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.99, (2, 1): 0.99, (2, 3): 0.99, (1, 4): 0.99, (4, 2): 10.0, (0, 3): 0.99, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.99, (3, 3): 0.99, (4, 1): 0.99, (3, 1): 0.99, (4, 4): 0.99, (2, 4): 0.99, (2, 0): -10.0, (4, 3): 0.99, (0, 4): 0.99, (3, 4): 0.99, (0, 2): 0.99}
=================> Iteration [2] <=================
{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 1.881, (2, 1): 1.8891, (2, 3): 1.8891, (1, 4): 1.881, (4, 2): 10.0, (0, 3): 1.881, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 1.881, (3, 3): 1.881, (4, 1): 9.179099999999998, (3, 1): 1.881, (4, 4): 1.881, (2, 4): 1.881, (2, 0): -10.0, (4, 3): 9.179099999999998, (0, 4): 1.881, (3, 4): 1.881, (0, 2): 1.881}
...
=================> Iteration [99] <=================
{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 9.527646095383496, (2, 1): 8.31246075943535, (2, 3): 9.9564283357929, (1, 4): 9.955342926058286, (4, 2): 10.0, (0, 3): 9.943610970197984, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 9.915379164837187, (3, 3): 9.978377732648427, (4, 1): 9.967856937611083, (3, 1): 9.540075053896025, (4, 4): 9.978377732648427, (2, 4): 9.96217538606548, (2, 0): -10.0, (4, 3): 9.988509945517464, (0, 4): 9.948995040592797, (3, 4): 9.970170640224508, (0, 2): 9.938818555890514}

=================> Iteration [100] <=================
{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 9.527646133852897, (2, 1): 8.31246079365578, (2, 3): 9.9564283357929, (1, 4): 9.955342926058286, (4, 2): 10.0, (0, 3): 9.943610970197984, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 9.915379166981248, (3, 3): 9.978377732648427, (4, 1): 9.96785693961782, (3, 1): 9.54007509002287, (4, 4): 9.978377732648427, (2, 4): 9.96217538606548, (2, 0): -10.0, (4, 3): 9.988509945517464, (0, 4): 9.948995040592797, (3, 4): 9.970170640224508, (0, 2):



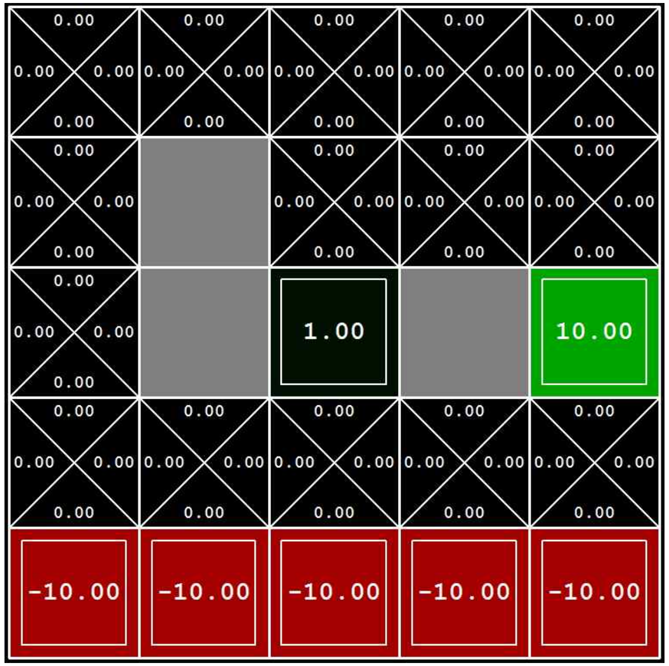
We selected a large discount and LivingReward with a small noise both to go distant exit and to avoid the cliff.

9.938818555890514}

Parameter values we found for Q3(e):
answerDiscount = 0.0, answerNoise = 0.0,
answerLivingReward = 0.0

```
================>   Iteration [ 1 ]   <================
{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.0, (2, 1): 0.0, (2, 3): 0.0, (1, 4): 0.0, (4, 2): 10.0, (0, 3): 0.0, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.0, (3, 3): 0.0, (4, 1): 0.0, (3, 1): 0.0, (4, 4): 0.0, (2, 4): 0.0, (2, 0): -10.0, (4, 3): 0.0, (0, 4): 0.0, (3, 4): 0.0, (0, 2): 0.0}

…

================>   Iteration [ 100 ]   <================
{(0, 0): -10.0, (3, 0): -10.0, (2, 2): 1.0, (1, 1): 0.0, (2, 1): 0.0, (2, 3): 0.0, (1, 4): 0.0, (4, 2): 10.0, (0, 3): 0.0, (1, 0): -10.0, (4, 0): -10.0, (0, 1): 0.0, (3, 3): 0.0, (4, 1): 0.0, (3, 1): 0.0, (4, 4): 0.0, (2, 4): 0.0, (2, 0): -10.0, (4, 3): 0.0, (0, 4): 0.0, (3, 4): 0.0, (0, 2): 0.0}
```

Q-Values after 100 Iterations for Q3(e)
[Avoid both exits (also avoiding the cliff)]



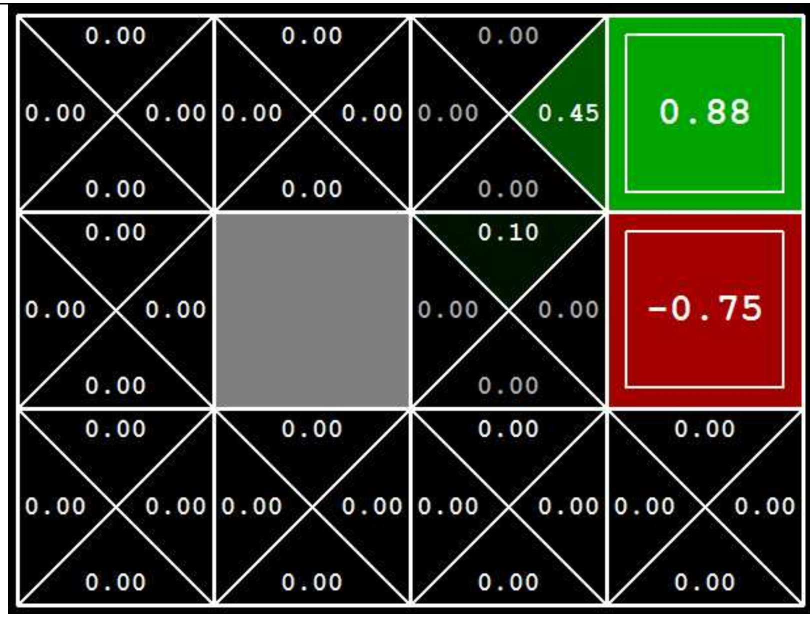
If all values are 0, then the iterations come to nothing.
This ends up with avoiding both exits for the agent.

Q-Learning result in Q4:

```
RUNNING 5 EPISODES

BEGINNING EPISODE: 1

…

Started in state: (3, 1)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: -1

EPISODE 1 COMPLETE: RETURN WAS -0.59049
```

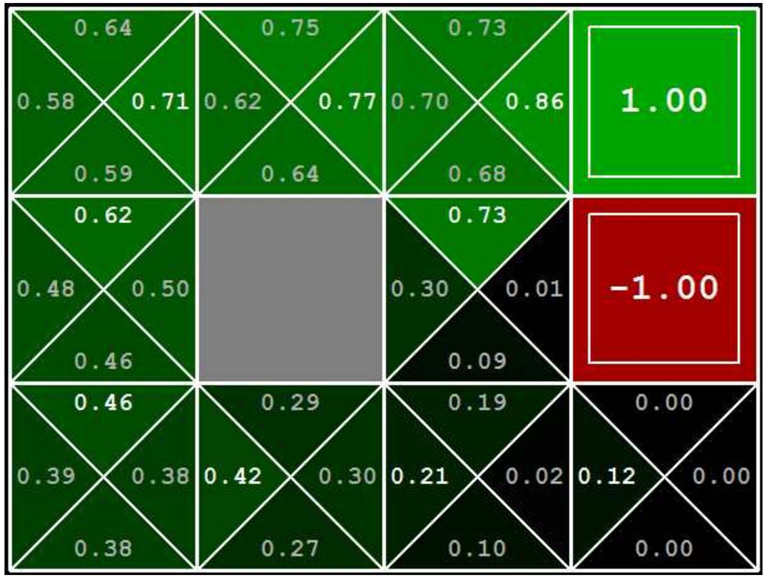
Q-Values after 5 Episodes manual moves for Q4

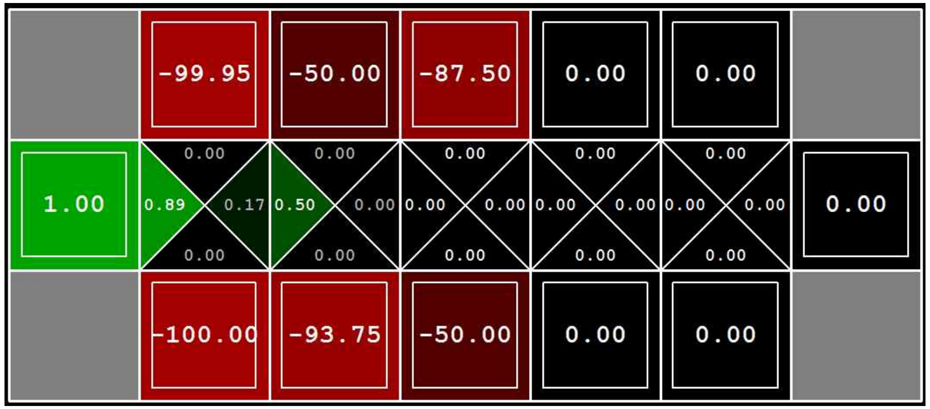
```
BEGINNING EPISODE: 2

…

Started in state: (3, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 1

EPISODE 2 COMPLETE: RETURN WAS 0.59049

BEGINNING EPISODE: 3

…

Started in state: (3, 1)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: -1

EPISODE 3 COMPLETE: RETURN WAS -0.6561

BEGINNING EPISODE: 4

…

Started in state: (3, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 1

EPISODE 4 COMPLETE: RETURN WAS 0.4782969

BEGINNING EPISODE: 5

…

Started in state: (3, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 1

EPISODE 5 COMPLETE: RETURN WAS 0.43046721
AVERAGE RETURNS FROM START STATE: 0.050532822
```



Q-Learning result in Q5:	**Q-Values after 100 Episodes for Q5**
	The "average returns" is lower than the q-values predict

<table>
<tr>
<td>EPISODE 100 COMPLETE: RETURN WAS 0.0984770902184
AVERAGE RETURNS FROM START STATE: 0.253883869937</td>
<td>because of the random actions and the initial learning phase.

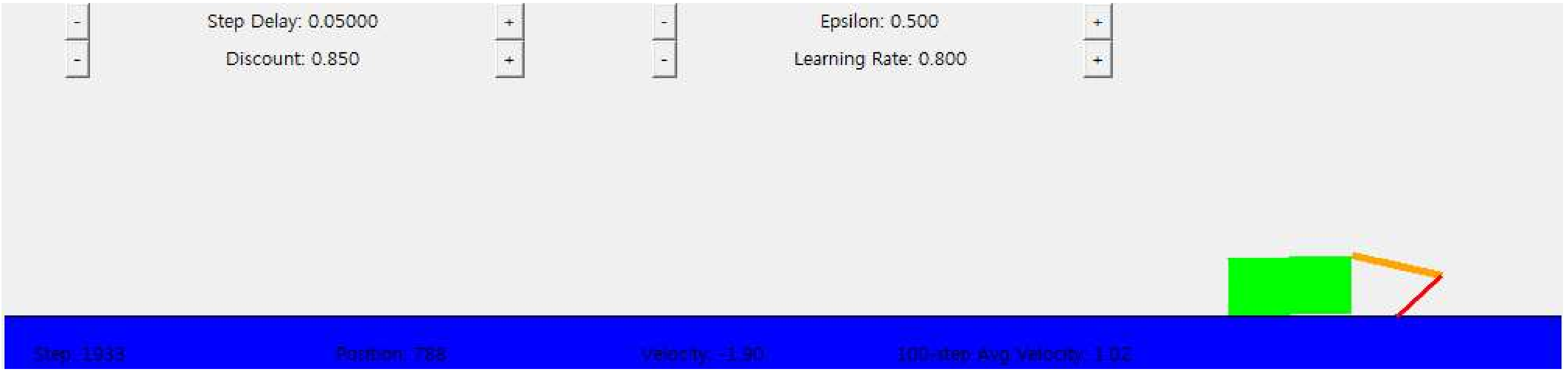


</td>
</tr>
<tr>
<td>Q-Learning result in Q6:</td>
<td>Parameters “-a q -k 50 -n 0 -g BridgeGrid -e 1”</td>
</tr>
<tr>
<td>EPISODE 50 COMPLETE: RETURN WAS 0.9
AVERAGE RETURNS FROM START STATE: -67.39542</td>
<td>

We think this is not possible because controlling epsilon and learning rate is not enough to go across the bridge with the constraint that the initial state is close to the beginning of the bridge and if the noise is fixed to zero (which means there is no chance of going to unintended direction)</td>
</tr>
</table>

Q-Learning result in Q7:

The crawling robot using our q-learner works quite well. We found it that at first this robot takes some time which motion is the best to go forward. Depending on the learning rate, the discount and the epsilon (randomly choosiness) value, the robot learns how to move in a different fashion. For example, if the epsilon holds high value then it gives many motions a try, whereas the low epsilon value has the robot keep the actions seen so far.



D. References

1. [Book] Machine_Learning by Tom_Mitchell

2. http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node41.html