Міністерство освіти і науки України

Національний технічний університет України «КПІ ім. Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Програмування інтелектуальних інформаційних систем

**ЗВІТ**

До лабораторної роботи №7.

Тема: Q-learning

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Виконав**  **студент** |  | ІТ-92, Бондаренко Дмитро Сергійович |  |  |
|  |  | (№ групи, прізвище, ім’я, по батькові ) |  |  |
|  |  |  |  |  |
| **Прийняв** |  | ас. Баришич Л.М. |  |  |
|  |  | (посада, прізвище, ім’я, по батькові ) |  |  |

Київ 2021

# **Завдання лабораторної роботи**

Завданням даної роботи є

1. Розробка ітеративного агента, що проходить заданий грід за допомогою q-learning.
2. Змінивши лише один параметри примусити агента перейти місток.
3. Навчитися керувати початковими політиками виконавши 5 завдань:
   1. Перейти до першого виходу (+1), ризикуючи шляхом поряд зі скелями (-10)
   2. Перейти до першого виходу (+1), оминаючи скелі (-10)
   3. Перейти до другого виходу (+10), ризикуючи шляхом поряд зі скелями (-10)
   4. Перейти до другого виходу (+10), оминаючи скелі (-10)
   5. Уникайте як виходів, так і скелі
4. Розробити Q-learning агента, реалізувавши епсилон-жадібний вибір дій, тобто він вибирає випадкові дії в епсилонній частині часу, а в іншому випадку слідує своїм поточним найкращим значенням Q.

# **Опис використаних технологій**

У цій роботі буде використана мова програмуванн Python, також буде використаний готовий проект із курсу Берклі, для написання завдань.

# **Опис програмного коду**

У роботі були дописані файли: valueIterationAgents.py та qlearningAgents.py де ми реалізували агентів.

valueIterationAgents.py

import util  
  
from learningAgents import ValueEstimationAgent  
  
  
class ValueIterationAgent(ValueEstimationAgent):  
 *"""  
 \* Please read learningAgents.py before reading this.\*  
  
 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, 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)  
 """* super().\_\_init\_\_()  
 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):  
 # Write value iteration code here  
 *"\*\*\* YOUR CODE HERE \*\*\*"* for i in range(self.iterations):  
 updatedValues = self.values.copy()  
  
 for state in self.mdp.getStates():  
  
 if self.mdp.isTerminal(state):  
 continue  
  
 actions = self.mdp.getPossibleActions(state)  
 optimal = max([self.getQValue(state, action) for action in actions])  
 updatedValues[state] = optimal  
  
 self.values = updatedValues  
  
 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 \*\*\*"  
  
 qval = 0  
  
 for s\_prime, T in self.mdp.getTransitionStatesAndProbs(state, action):  
 qval += T \* (self.mdp.getReward(state, action, s\_prime) + self.discount \* self.getValue(s\_prime))  
  
 return qval  
  
 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.  
 """* "\*\*\* YOUR CODE HERE \*\*\*"  
  
 policy = util.Counter()  
  
 for action in self.mdp.getPossibleActions(state):  
 policy[action] = self.getQValue(state, action)  
  
 return policy.argMax()  
  
 def getPolicy(self, state):  
 return self.computeActionFromValues(state)  
  
 def getAction(self, state):  
 *"Returns the policy at the state (no exploration)."* return self.computeActionFromValues(state)  
  
 def getQValue(self, state, action):  
 return self.computeQValueFromValues(state, action)

qlearningAgents.py

from learningAgents import ReinforcementAgent  
  
import random, util  
  
class QLearningAgent(ReinforcementAgent):  
 *"""  
 Q-Learning Agent  
  
 Functions you should fill in:  
 - computeValueFromQValues  
 - computeActionFromQValues  
 - getQValue  
 - getAction  
 - 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.values = util.Counter()  
  
 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  
 """* "\*\*\* YOUR CODE HERE \*\*\*"  
  
 return self.values[(state,action)]  
  
  
  
  
 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.  
 """* "\*\*\* YOUR CODE HERE \*\*\*"  
  
 Qval = float('-inf')  
  
 for action in self.getLegalActions(state):  
 Qval = max(self.getQValue(state, action), Qval)  
  
 if Qval == float('-inf'):  
 return 0.0  
 else : return Qval  
  
  
 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.  
 """* "\*\*\* YOUR CODE HERE \*\*\*"  
  
 if len(self.getLegalActions(state)) is 0:  
 return None  
  
 val = self.computeValueFromQValues(state)  
 actions = []  
  
 for action in self.getLegalActions(state):  
 if val is self.getQValue(state, action):  
 actions.append(action)  
  
 return random.choice(actions)  
  
  
 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 util.flipCoin(self.epsilon):  
 action = random.choice(legalActions)  
 else:  
 action = self.computeActionFromQValues(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 \*\*\*"  
  
 self.values[(state, action)] = (1-self.alpha) \* self.values[(state,action)] + self.alpha \* (reward + self.discount\*self.computeValueFromQValues(nextState))  
  
  
 def getPolicy(self, state):  
 return self.computeActionFromQValues(state)  
  
 def getValue(self, state):  
 return self.computeValueFromQValues(state)

# **Скріншоти роботи програмного застосунку**

Завдання 1:



Завдання 2:

Для того, щоб зробити це завдання, нам треба змінити деякі параметри, які ми передаємо як аргументи консолі. Розберемо кожен із цих параметрів, а також параметри, які будуть використовуватися у майбутніх завданнях.

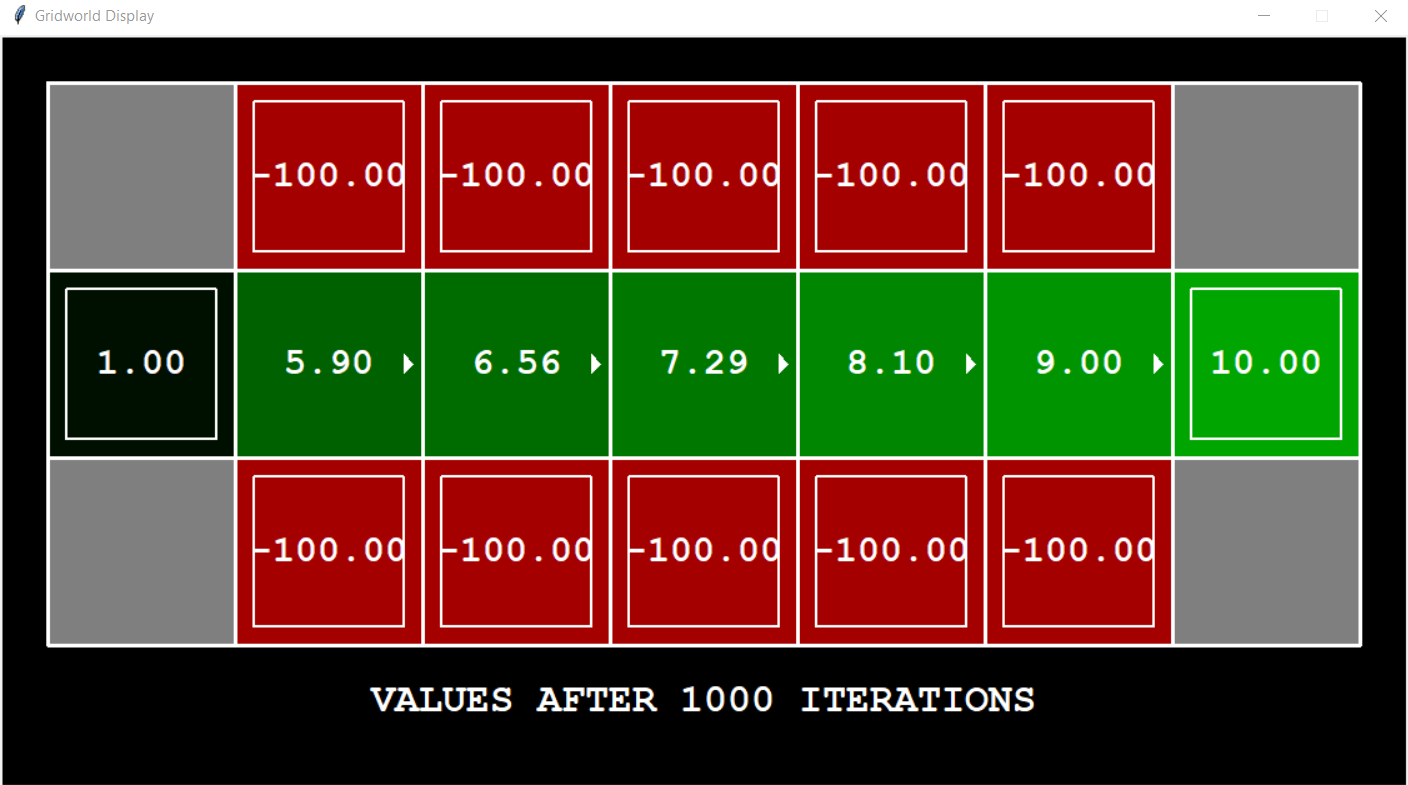
--discount – використовується для балансування негайної та майбутньої винагороди.

--noise – як часто дії призводять до ненавмисного напрямку.

--epsilon - можливість виконання випадкових дій у Q-learning

--livingReward – винагорода за життя за крок у часі

Для того, щоб агент міг перейти через міст, ми маємо змінити параметр –noise в 0.

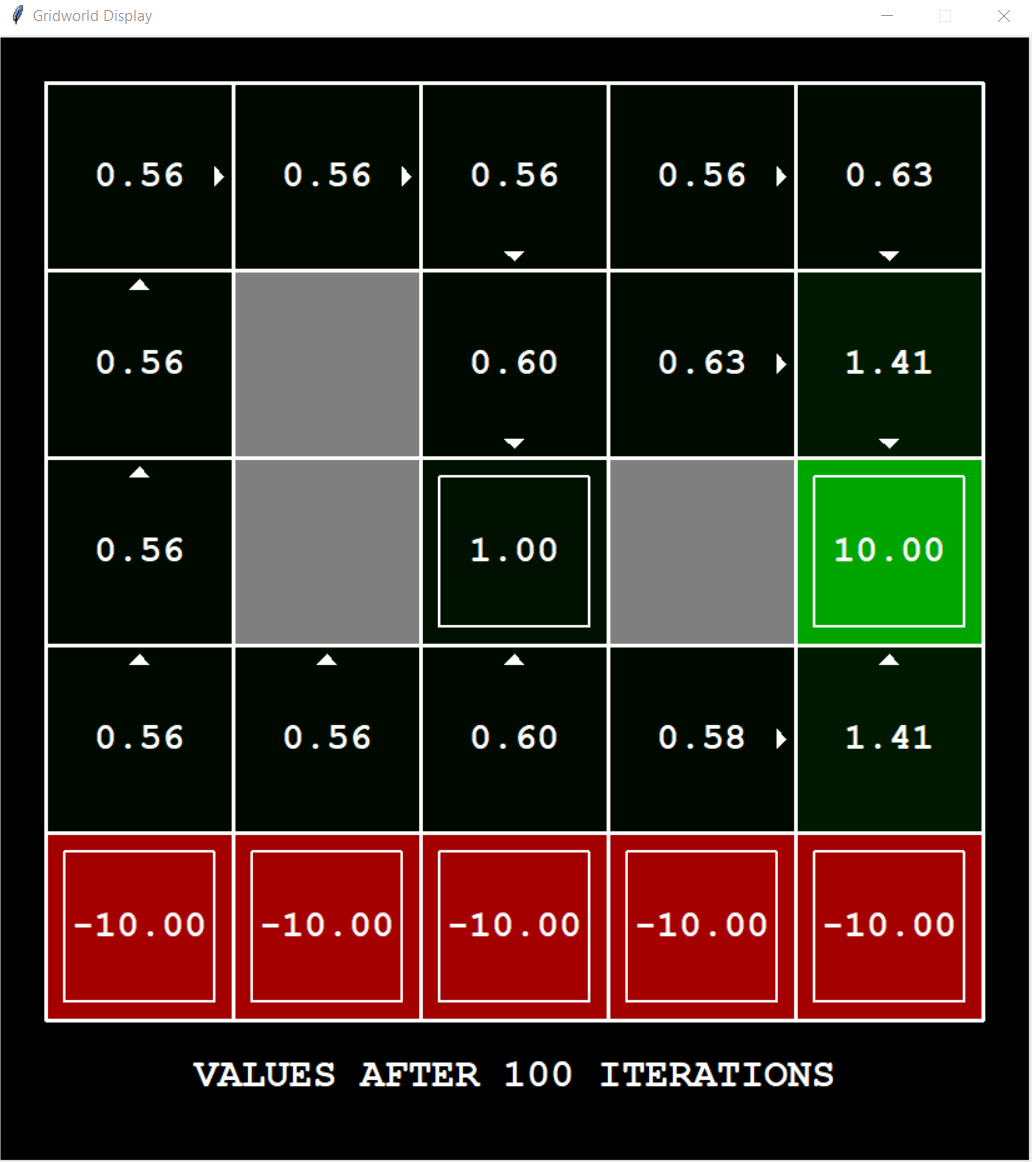


Завдання 3:

Для виконання цього завдання ми будемо маніпулювати такими параметрами як --discount, --noise, --livingReward.

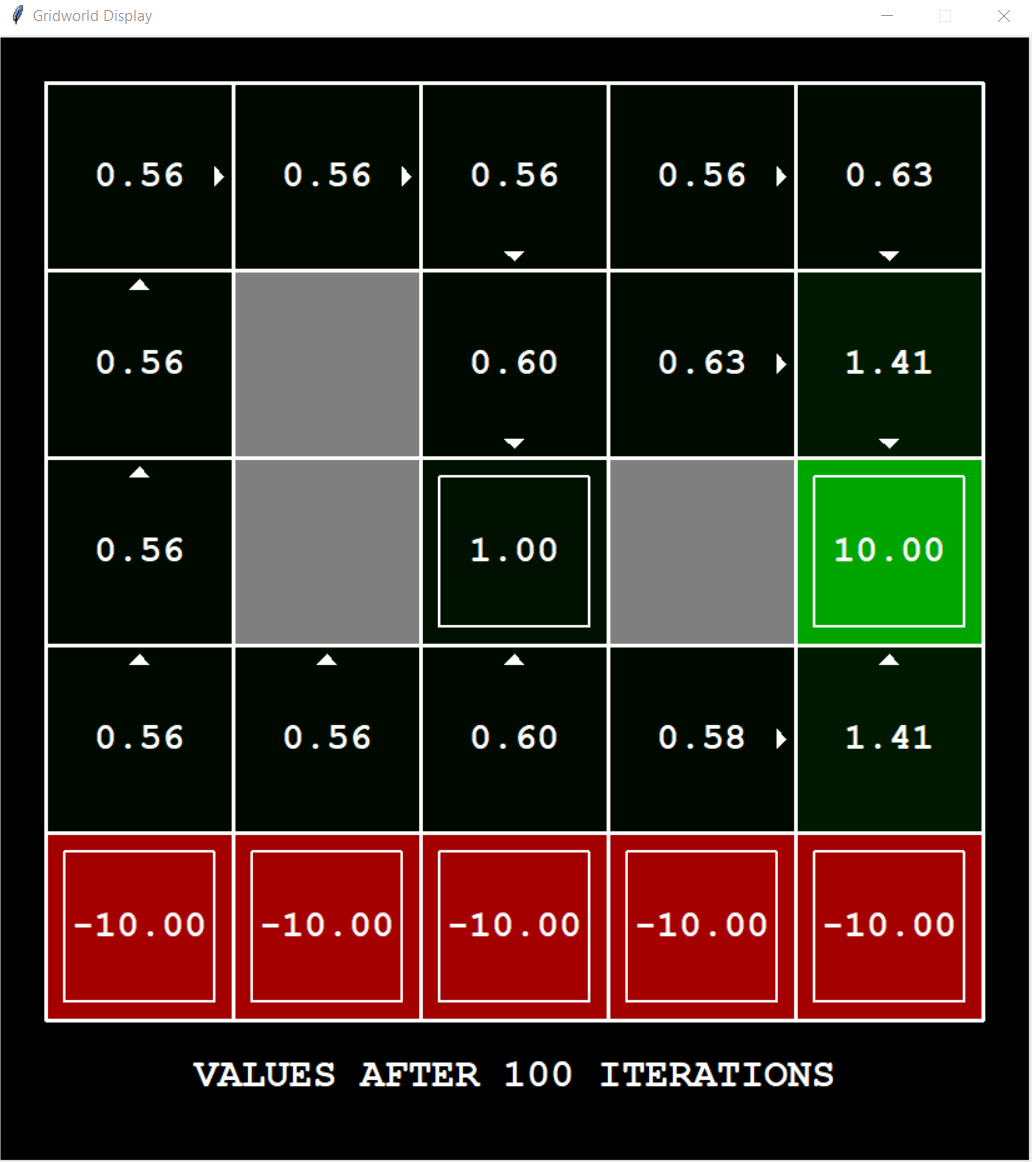
1. Перейти до першого виходу (+1), ризикуючи шляхом поряд зі скелями (-10)

Параметри --discount: 0.1, --noise: 0, --livingReward: 0.5



1. Перейти до першого виходу (+1), оминаючи скелі (-10)

Параметри --discount: 0.1, --noise: 0.1, --livingReward: 0.5



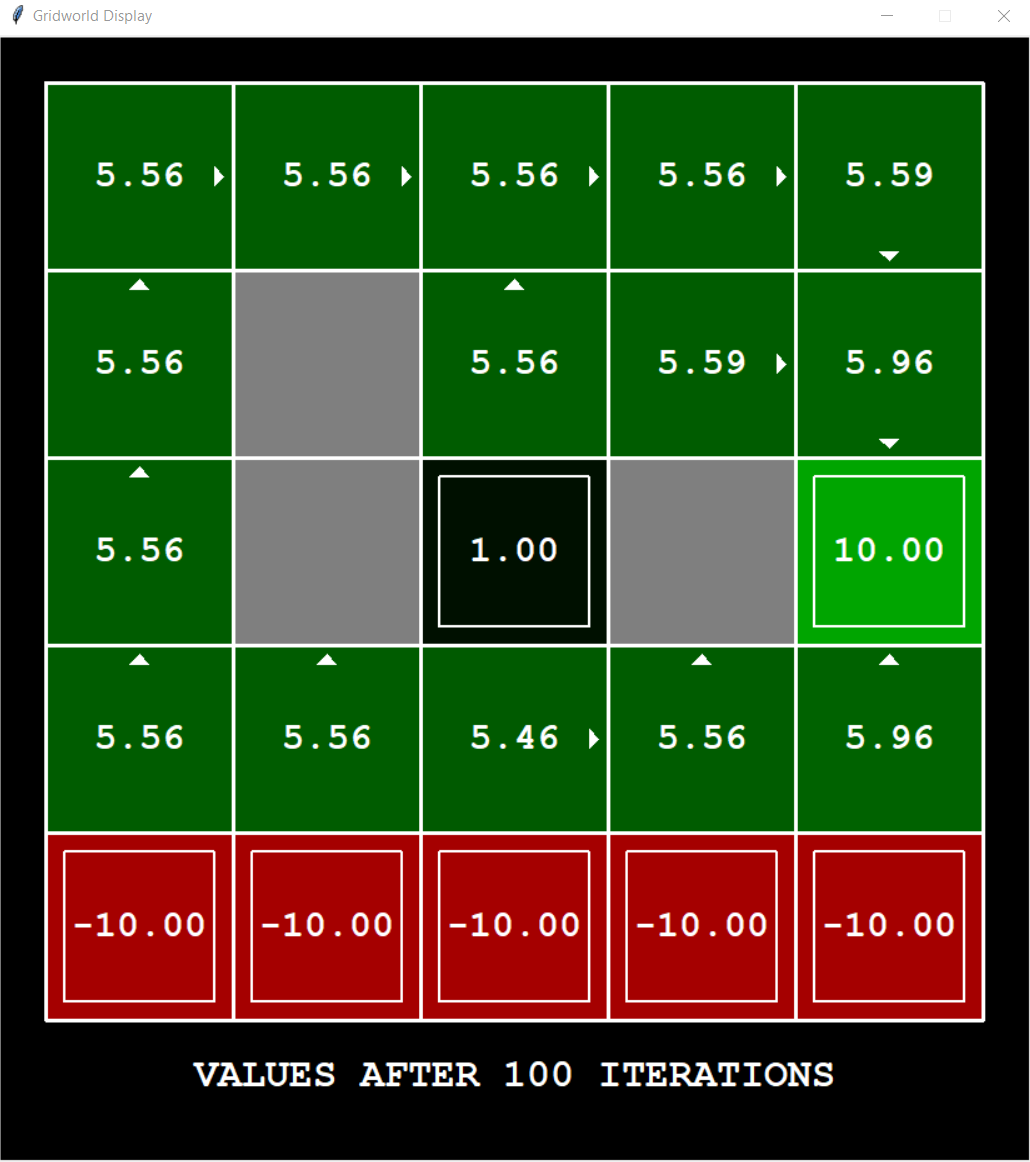
1. Перейти до другого виходу (+10), ризикуючи шляхом поряд зі скелями (-10)

Параметри --discount: 0.1, --noise: 0, --livingReward: 5



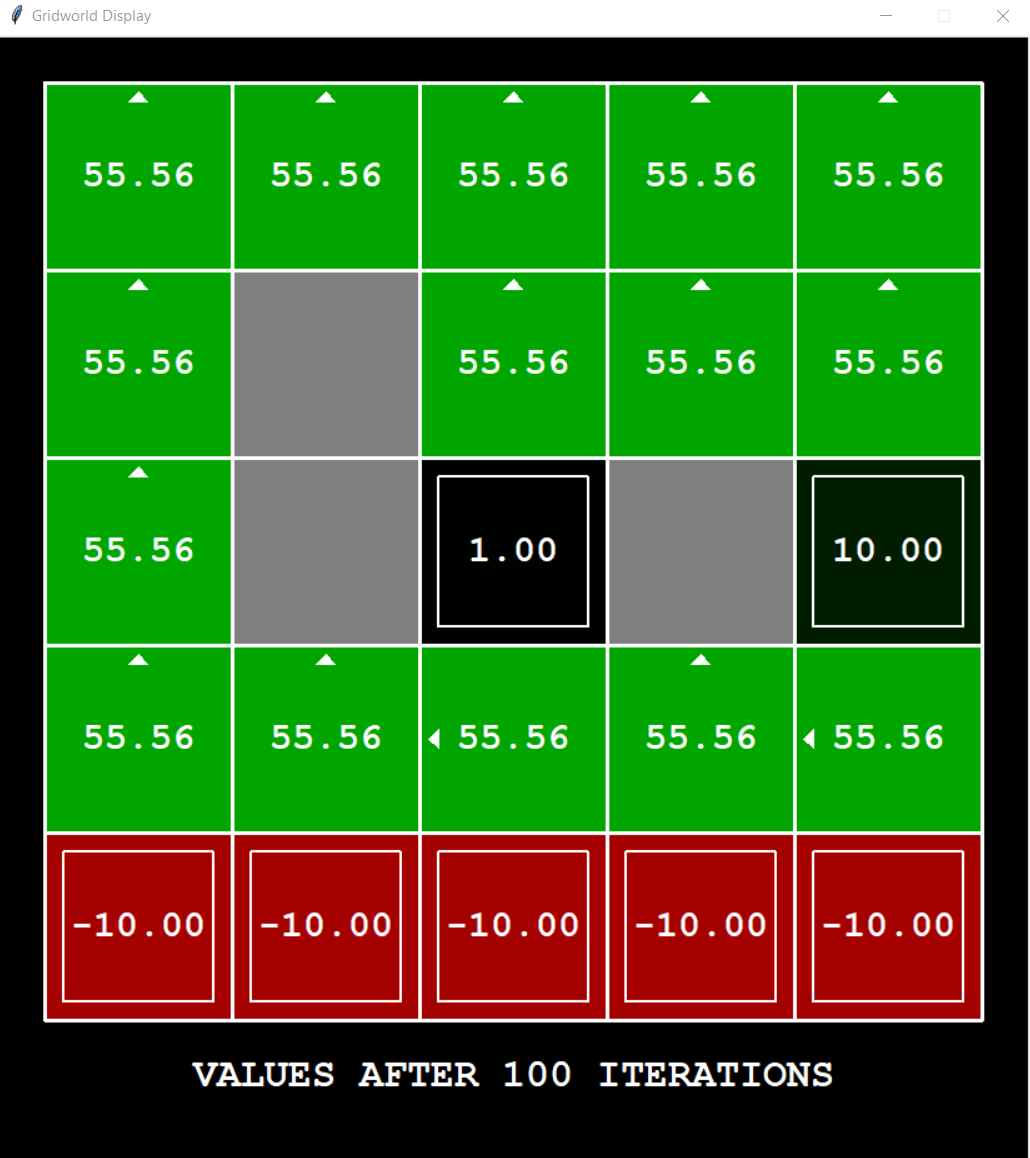
1. Перейти до другого виходу (+10), оминаючи скелі (-10)

Параметри --discount: 0.1, --noise: 0.1, --livingReward: 5



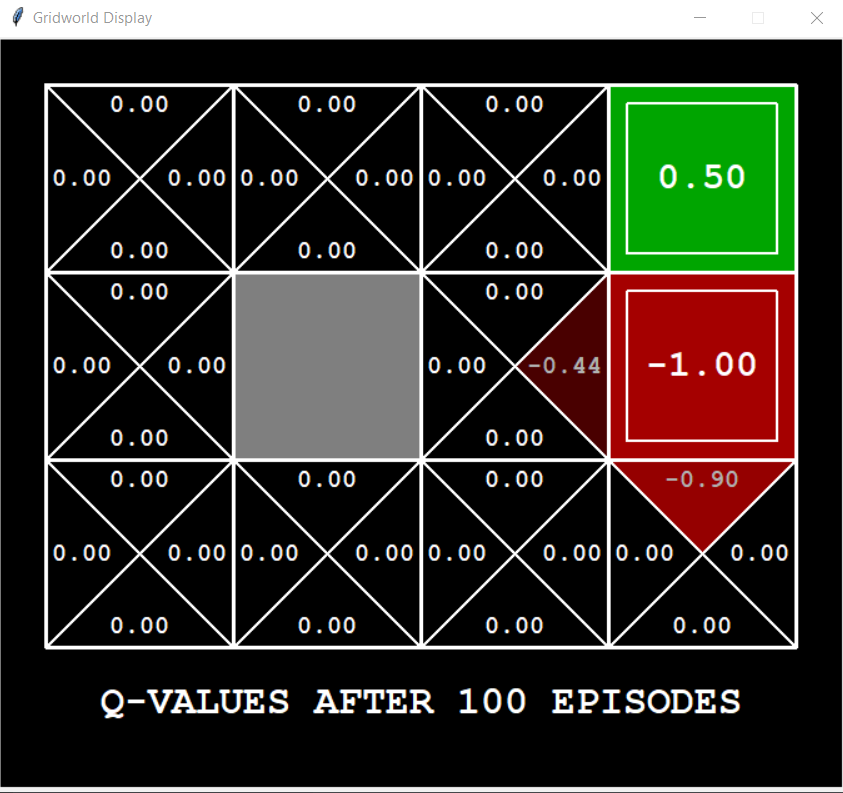
1. Уникайте як виходів, так і скелі

Параметри --discount: 0.1, --noise: 0, --livingReward: 50

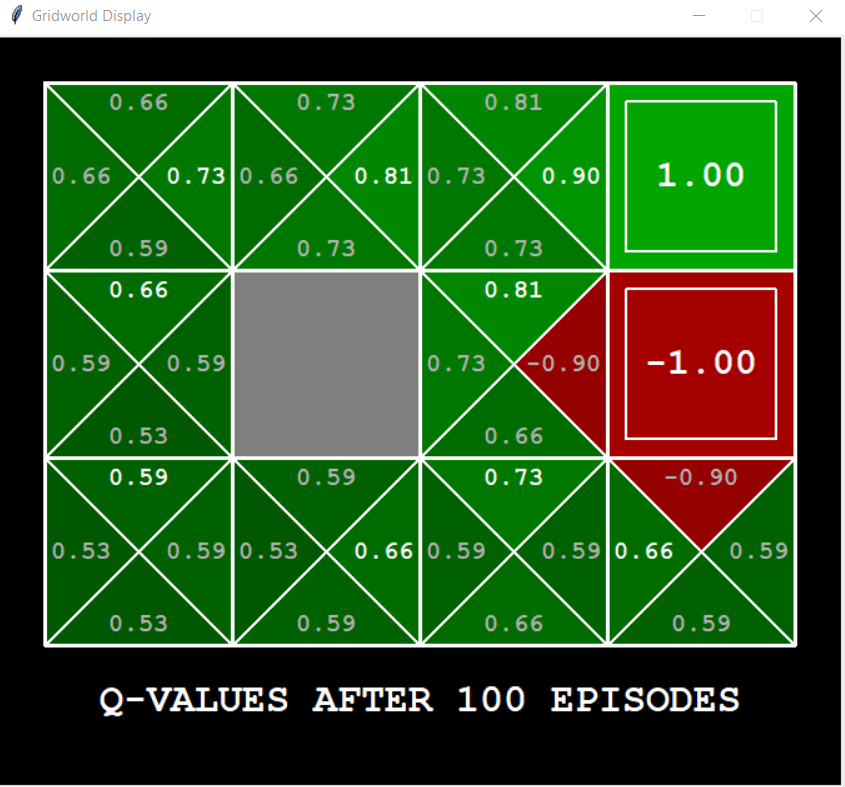


Завдання 4:

--epsilon: 0.1



--epsilon: 0.9



**Висновок:** У цій роботі ми навчилися будувати моделі для QL навчилися керувати початковими політиками, побудували агента QL, який працює в залежності від епсілон значення.