

ЛАБОРАТОРНАЯ РАБОТА №5

«ИССЛЕДОВАНИЕ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ»

1. Цель работы

Исследование методов недетерминированного поиска решений задач, приобретение навыков программирования интеллектуальных агентов, функционирующих в недетерминированных средах, исследование методов построения агентов, обучаемых на основе алгоритмов обучения с подкреплением.

2. Постановка задачи

2.1. Изучить по лекционному материалу и учебным пособиям [1-3, 7] методы недетерминированного поиска, основанные на использовании модели марковского процесса принятия решений, включая алгоритмы итерации по значениям и политикам, а также алгоритмы обучения подкреплением с моделями и без моделей: алгоритм прямого обучения, алгоритм TD-обучения, алгоритм Q-обучения.

2.2. Использовать для выполнения лабораторной работы файлы из архива МИСИИ_лаб_5.zip. Разверните программный код в новой папке и не смешивайте с файлами предыдущих лабораторных работ.

2.3. В этой лабораторной работе необходимо реализовать алгоритмы итераций по значениям и Q-обучение. Сначала протестируете функционирование агентов в клеточном мире (класс Gridworld), а затем используйте их для моделирования работа Crawler и игры Пакман.

В состав файлов лабораторной работы входит автооценщик, с помощью которого вы можете оценивать свои решения. Для автооценки всех заданий лабораторной работы следует выполнить команду:

```
python autograder.py
```

Для оценки конкретного задания, например, задания 2, автооценщик вызывается с параметром q2, где 2 – номер задания:

```
python autograder.py -q q2
```

Для проверки отдельного теста в пределах задания используйте команды вида:

```
python autograder.py -t test_cases/q2/1-bridge-grid
```

2.4. Для начала запустите среду Gridworld в ручном режиме, в котором для управления используются клавиши со стрелками:

```
python gridworld.py -m
```

Вы увидите среду Gridworld в виде клеточного мира размером 4x3 с двумя терминальными состояниями. Синяя точка — это агент. Обратите внимание, что, когда вы нажимаете клавишу «вверх», агент фактически перемещается на Север только в 80% случаев. Это свойство агента в среде Gridworld. Вы можете контролировать многие опции среды Gridworld. Полный список опций можно получить по команде:

```
python gridworld.py -h
```

Агент по умолчанию перемещается по клеткам случайным образом. При выполнении команды

```
python gridworld.py -g MazeGrid
```

Вы увидите, как агент случайно перемещается по клеткам, пока не попадет в клетку с терминальным состоянием. Не самый звездный час для такого агента (низкое вознаграждение).

Примечание: среда Gridworld такова, что вы сначала должны войти в предтерминальное состояние (поля с двойными линиями, показанные в графическом интерфейсе), а затем выполнить специальное действие «выход» до фактического завершения эпизода (в истинном терминальном состоянии,

называемом `TERMINAL_STATE`, которое не отображается в графическом интерфейсе). Если вы выполните выход вручную, накопленное вознаграждение может быть меньше, чем вы ожидаете, из-за дисконтирования (опция `-d`; по умолчанию коэффициент дисконтирования равен 0,9). Просмотрите результаты, которые выводятся в консоли. Вы увидите сведения о каждом переходе, выполняемом агентом.

Как и в `Rastan`, позиции представлены декартовыми координатами (x , y), а любые массивы индексируются $[x][y]$, где «север» является направлением увеличения y и т. д. По умолчанию большинство переходов получают нулевую награду. Это можно изменить с помощью опции `-r`, которая управляет текущей наградой.

2.5. В задании 1 требуется реализовать следующие методы класса `ValueIterationAgent(ValueEstimationAgent)`:

- `runValueIteration(self)`;
- `computeQValueFromValues(self, state, action)`;
- `computeActionFromValues(self, state)`.

Метод `runValueIteration(self)` для заданного числа итераций и для каждого состояния `state` осуществляет выбор действия в состоянии и вычисляет значения q -ценностей и складывает их в списке `q_state` с помощью вызова `q_state.append(self.getQValue(state, action))`. Вычисление ценности каждого состояния реализуется на основе (5.9) путем вызова `updatedValues[state] = max(q_state)`, где `updatedValues` – временный словарь, содержащий обновленные ценности состояний `state` на каждой итерации. После вычисления ценности всех состояний на заданной итерации они сохраняются в словаре значений ценности состояний объекта с помощью вызова `self.values = updatedValues`.

Метод `computeQValueFromValues(self, state, action)` вычисляет ценности q -состояний в соответствии с (5.10). Для каждого следующего состояния `nextstate` после `state` вычисления реализуются с помощью вызова:

$Qvalue += prob * (self.mdp.getReward(state, action, nextstate) + self.discount * self.getValue(nextstate))$, где `nextstate` и `prob` получают путем операции `in` для множества `self.mdp.getTransitionStatesAndProbs(state, action)`.

Метод `computeActionFromValues(self, state)` определяет лучшее действие в состоянии `state`. Для этого он перебирает все доступные действия `action` в состоянии `state`, получаемые с помощью вызова `self.mdp.getPossibleActions(state)` и для каждого действия `action` вычисляет и запоминает q -ценности в словаре `policy` путем вызова `policy[action] = self.getQValue(state, action)`. В заключение метод возвращает лучшее действие путем вызова `policy.argmax()`, что соответствует извлечению политики согласно (5.15).

2.6. В задании 2 требуется решить задачу прохождения агентом моста `BridgeGrid`. В задании используется агент, выполняющий итерации по значениям и реализованный в задании 1. Для решения задачи необходимо определить корректные значения только одного из двух параметров агента: `answerDiscount` – коэффициента дисконтирования или `answerNoise` – уровня шума и указать в функции `question2()` в файле `analysis.py`. По умолчанию эти параметры имеют значения `answerDiscount = 0.9`, `answerNoise = 0.2`. Решение здесь тривиально – агент не должен иметь возможности отклоняться от прямого маршрута из начального состояния в конечное.

2.7. В задании 3 необходимо подобрать значения коэффициента дисконтирования (`answerDiscount`), уровня шума (`answerNoise`) и текущей награды (`answerLivingReward`) для среды `DiscountGrid` (рисунок 5.4). Помните, что с помощью значений текущей награды можно управлять степенью риска: большие положительные награды заставляют агента избегать выхода, умеренные положительные награды исключают риск, умеренные отрицательные допускают риск.

Снижение уровня шума понижает вероятность отклонения от выбранного направления движения. Коэффициент дисконтирования определяет важность ближайших наград по отношению к будущим наградам.

2.8. В задании 4 необходимо реализовать асинхронную итерацию по значениям ценности состояний. При выполнении задания необходимо переопределить единственный метод, `runValueIteration` класса `AsynchronousValueIterationAgent`, который является наследником класса `ValueIterationAgent`.

В случае асинхронной итерации по значениям на каждой итерации обновляется ценность только одного состояния, в отличие от выполнения пакетного обновления, когда обновляются все состояния. Указанный процесс выбора состояния `state` для обновления на итерации i можно реализовать через индексацию списка состояний: `state = states[i % num_states]`, где `num_states` - число состояний, а `states` – список состояний, возвращаемый вызовом `states = self.mdp.getStates()`. При этом обновление ценности состояния выполняется по-прежнему на основе уравнений (5.9) и (5.10). Для этого для выделенного состояния `state` перебираем возможные действия `action` и вычисляем ценности q -состояний с помощью метода `self.computeQValueFromValues(state, action)` родительского класса и для каждого состояния выбираем действие, обеспечивающие максимальную q -ценность `max_val`. В результате сохраняем максимальную ценность в словаре состояний объекта `self.values[state] = max_val`.

2.9. В задании 5 необходимо реализовать алгоритм итераций по приоритетным значениям. При реализации алгоритма следуйте указаниям, сформулированным в самом задании. Поиск с итерациями по приоритетным значениям фокусируется на обновлении значений состояний, которые более вероятно изменяют политику.

Идея заключается в обновлении тех состояния, для которых изменения более ожидаемы. Если разность большая, то следует в приоритетном порядке обновить ценность предшествующих состояний s . Соответственно для этого формируется очередь из состояний, которая упорядочивается в соответствии со значениями приращения (разности) ценности состояний.

2.10. В задании 6 необходимо реализовать методы `update`, `computeValueFromQValues`, `getQValue` и `computeActionFromQValues`.

Реализация метода `getQValue(self, state, action)` тривиальна. Метод просто обращается к свойству класса `self.values[(state, action)]` и получает значения $q(s,a)$.

При реализации метода `computeValueFromQValues(self, state)` необходимо для всех легальных действий `action` в состоянии `state` вычислить ценности q -состояний с помощью `getQValue(state, action)` и вернуть максимальную ценность. Реализация метода `computeActionFromQValue self, state)` использует вызов `bestQ=computeValueFromQValues(self, state)` для получения значения оптимальной ценности `bestQ`. Затем в цикле перебираются все допустимые действия для `state`, вычисляются с помощью `getQValue(state, action)` ценности q -состояний и определяются действия, соответствующие оптимальной ценности `bestQ`. Метод осуществляет случайный выбор лучшего действия среди найденных лучших действий.

В методе `update(self, state, action, nextState, reward)` необходимо реализовать q -обучение в соответствии с (5.23) и (5.24). Для вычисления ценности состояния `nextState` используйте вызов `getValue(nextState)`, который реализует (5.9).

2.11. В задании 7 необходимо реализовать эпсилон-жадную стратегию в методе `getAction(self, state)`. Реализация метода предполагает, что подбрасывается монетка с помощью вызова метода `util.flipCoin(self.epsilon)` и с вероятностью `epsilon` возвращается случайное из допустимых действий в состоянии `state` иначе возвращается лучшее действие, определяемое с помощью `computeActionFromQValue self, state)`.

2.12. При выполнении задания 8 следуйте указаниям, приведенным в самом задании. Проведите все необходимые эксперименты, указанные в задании.

2.13. При выполнении задания 9 следуйте указаниям, приведенным в самом задании. Проведите все необходимые варианты игры Пакман, указанные в задании.

2.14. В задании 10 необходимо реализовать q-обучение с аппроксимацией.

При написании кода метода `getQValue(self, state, action)`, возвращающего аппроксимированное значение $Q(s,a)$, необходимо получить вектор признаков q-состояний с помощью вызова `self.features = self.featExtractor.getFeatures(state, action)`. А затем для всех признаков i из `self.features[i]` найти взвешенную сумму признаков в соответствии с (5.25).

Реализация метода `update(self, state, action, nextState, reward)` должна обеспечивать вычисление обновления весов признаков. Для этого вычисляется значение `difference` - разности выборочной и ожидаемой оценок $Q(s, a)$, затем для каждого признака `self.features[i]` вычисляются обновления весов `self.weights[i]` в соответствии с (5.26).

3. Ход работы

Задание 1 (4 балла). Итерации по значениям

Реализуйте агента, осуществляющего итерации по значениям в соответствии с выражением (5.13), в классе `ValueIterationAgent` (класс частично определен в файле `valueIterationAgents.py`). Агент `ValueIterationAgent` получает на вход MDP при вызове конструктора класса и выполняет итерации по значениям для заданного количества итераций (опция `-i`) до выхода из конструктора.

При итерации по значениям вычисляются k -шаговые оценки оптимальных значений V_k . Дополнительно к `runValueIteration`, реализуйте следующие методы для класса `ValueIterationAgent`, используя значения V_k :

`computeActionFromValues(state)` - определяет лучшее действие в состоянии (политику) с учетом значений ценности состояний, хранящихся в словаре `self.values`;

`computeQValueFromValues(state, action)` возвращает Q-ценность пары `(state, action)` с учетом значений ценности состояний, хранящихся в словаре `self.values` (данный метод реализует вычисления с учетом уравнения Беллмана для q-ценностей);

Вычисленные значения отображаются в графическом интерфейсе пользователя (см. рисунки ниже): ценности состояний представляются числами в квадратах, значения Q - ценностей отображаются числами в четвертях квадратов, а политики — это стрелки, исходящие из каждого квадрата.

Важно: используйте «пакетную» версию итерации по значениям, где каждый вектор ценности состояний V_k вычисляется на основе предыдущих значений вектора V_{k-1} , а не на основе «онлайн» версии, где один и тот же вектор обновляется по месту расположения. Это означает, что при обновлении значения состояния на итерации k на основе значений его состояний-преемников, значения состояния-преемника, используемые в вычислении обновления, должны быть значениями из итерации $k-1$ (даже если некоторые из состояний-преемников уже были обновлены на итерации k).

Примечание: политика, сформированная по значениям глубины k , фактически будет соответствовать следующему значению накопленной награды (т.е. вы вернете π_{k+1}). Точно так же Q-ценности дают следующее значение награды (т.е. вы вернете Q_{k+1}). Вы должны вернуть политику π_{k+1} .

Подсказка: при желании вы можете использовать класс `util.Counter` в `util.py`, который представляет собой словарь ценностей состояний со значением по умолчанию, равным нулю. Однако будьте осторожны с `argMax`: фактический `argmax`, который вам необходим, может не быть ключом!

Примечание. Обязательно обработайте случай, когда состояние не имеет доступных действий в MDP (подумайте, что это означает для будущих вознаграждений).

Чтобы протестировать вашу реализацию, запустите автооценщик:

python autograder.py -q q1

python gridworld.py -a value -i 5

Реализация

В листинге 1 представлен код реализации класса ValueIterationAgent в файле valueIterationAgent.py

Листинг 1 – Код класса ValueIterationAgent

```
class ValueIterationAgent(ValueEstimationAgent):
    """
    * Пожалуйста, прочтите learningAgents.py перед тем, как читать
    это. *

    ValueIterationAgent принимает марковский процесс принятия
    решений
    (см. mdp.py) при инициализации и выполняет итерацию по значениям
    для заданного количества итераций с использованием
    коэффициента дисконтирования.

    """
    def __init__(self, mdp, discount = 0.9, iterations = 100):
        """
        Ваш агент итераций по значениям должен принимать mdp при
        вызове конструктора, запустите его с указанным количеством
        итераций,
        а затем действуйте в соответствии с полученной политикой.

        Некоторые полезные методы mdp, которые вы будете использовать:
            mdp.getStates() - возвращает список состояний MDP
            mdp.getPossibleActions(state) - возвращает кортеж возможных
            действий в состоянии
            mdp.getTransitionStatesAndProbs(state, action) - возвращает
            список из пар (nextState, prob) - s' и вероятности переходов T(s,a,s')
            mdp.getReward(state, action, nextState) - возвращает награду
            R(s,a,s')
```

```

        mdp.isTerminal(state) - проверяет, является ли состояние
терминальным
    """
    self.mdp = mdp
    self.discount = discount
    self.iterations = iterations
    self.values = util.Counter() # Counter - словарь ценностей
состояний, по умолчанию содержит 0
    self.runValueIteration()

def runValueIteration(self):
    # Напишите код, реализующий итерации по значениям
    """** ВСТАВЬТЕ ВАШ КОД СЮДА """
    states = self.mdp.getStates()

    for i in range(self.iterations):
        temp = util.Counter()

        for state in states:
            best = float("-inf")
            actions = self.mdp.getPossibleActions(state)

            for action in actions:
                transitions = self.mdp.getTransitionStatesAndProbs(state,
action)

                sum = 0

                for transition in transitions:
                    reward = self.mdp.getReward(state, action, transition[0])
                    sum += transition[1]*(reward +
self.discount*self.values[transition[0]])

                best = max(best, sum)

            if best != float("-inf"):
                temp[state] = best

        for state in states:
            self.values[state] = temp[state]

def getValue(self, state):
    """

```

```

        Возвращает ценность состояния (вычисляется в __init__).
        """
        return self.values[state]

def computeQValueFromValues(self, state, action):
    """
        Вычисляет Q-ценность в состоянии по
        значению ценности состояния, сохраненному в self.values.
        """

    transitions = self.mdp.getTransitionStatesAndProbs(state,
action)

    sum = 0

    for transition in transitions:
        reward = self.mdp.getReward(state, action, transition[0])
        sum += transition[1]*(reward +
self.discount*self.values[transition[0]])

    return sum

def computeActionFromValues(self, state):
    """
        Определяет политику - лучшее действие в состоянии
        в соответствии с ценностями, хранящимися в self.values.

        Обратите внимание, что если нет никаких допустимых действий,
        как в случае терминального состояния, необходимо вернуть None.
        """
    """*** ВСТАВЬТЕ ВАШ КОД СЮДА ***"""

    best = float("-inf")
    act = None
    actions = self.mdp.getPossibleActions(state)
    for action in actions:
        q = self.computeQValueFromValues(state, action)
        if q > best:
            best = q
            act = action

    return act

```

```

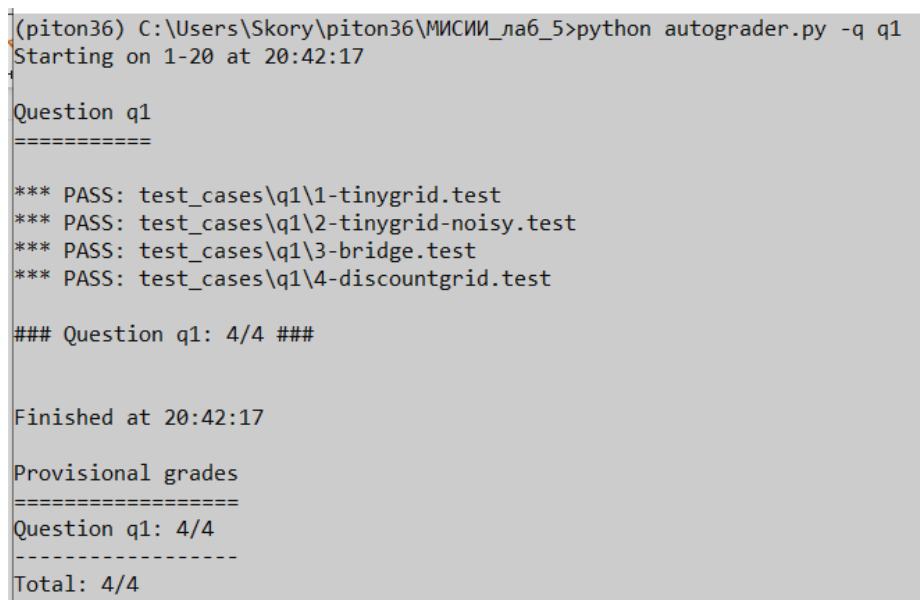
def getPolicy(self, state):
    return self.computeActionFromValues(state)

def getAction(self, state):
    "Возвращает политику в состоянии (без исследования)"
    return self.computeActionFromValues(state)

def getQValue(self, state, action):
    return self.computeQValueFromValues(state, action)

```

Разработанный код был проверен с помощью автооценивания. Результаты представлены на рисунке 1.



```

(python36) C:\Users\Skory\python36\МИСИИ_лаб_5>python autograder.py -q q1
Starting on 1-20 at 20:42:17

Question q1
=====

*** PASS: test_cases\q1\1-tinygrid.test
*** PASS: test_cases\q1\2-tinygrid-noisy.test
*** PASS: test_cases\q1\3-bridge.test
*** PASS: test_cases\q1\4-discountgrid.test

### Question q1: 4/4 ###

Finished at 20:42:17

Provisional grades
=====
Question q1: 4/4
-----
Total: 4/4

```

Рисунок 1 – Результаты тестирования кода для задания 1

В среде BookGrid, используемой по умолчанию, при выполнении 5 итераций по значениям должен получиться результат, изображенный на рисунке 5.3 методических указаний. Была введена команда: `python gridworld.py -a value -i 5`

Результат работы команды изображен на рисунке 2. Как видно, результаты совпадают, значит, код написан верно.



Рисунок 2 – Результат при выполнении 5 итераций

Задание 2 (1 балл). Анализ перехода через мост

BridgeGrid — это клеточный мир с высоким вознаграждением в целевом конечном состоянии, к которому ведет узкий «мост», по обе стороны от которого находится пропасть с высоким отрицательным вознаграждением.

Агент начинает движение из состояния с низким вознаграждением. С коэффициентом дисконтирования, по умолчанию равным 0,9, и уровнем шума, по умолчанию равным 0.2, оптимальная политика не обеспечивает прохождения моста. Измените только ОДИН из параметров - коэффициент дисконтирования или уровень шума, чтобы при оптимальной политике агент попытался пересечь мост.

Поместите свой ответ в код функции question2() в файле analysis.py. (Шум соответствует тому, как часто агент попадает в незапланированное состояние-преемник при выполнении действия). Значения по умолчанию соответствуют вызову:

```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

Оценивание: автооценщик проверяет, чтобы вы изменили только один из указанных параметров, и что при этом изменении корректный агент итераций по значениям пересекает мост. Чтобы проверить свой ответ, запустите автооценщик:

`python autograder.py -q q2`

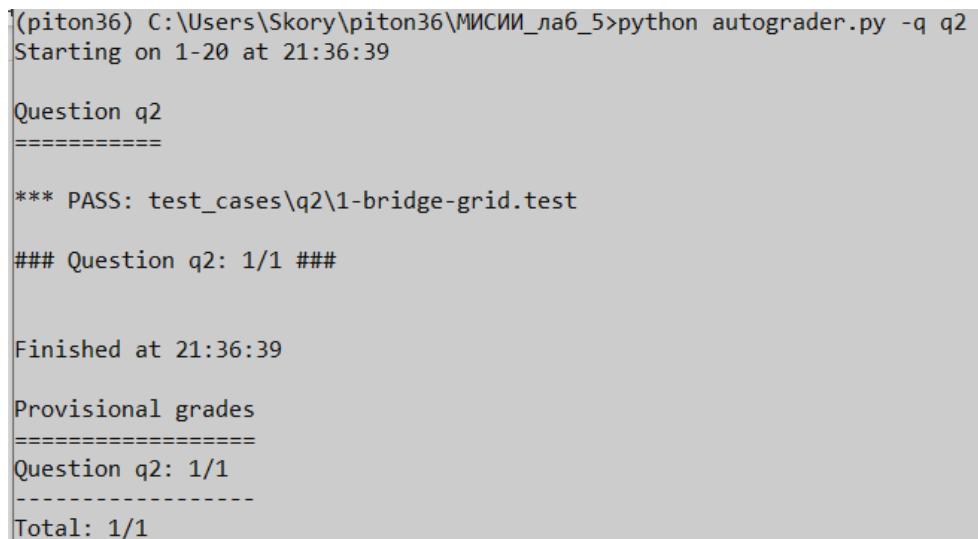
Реализация

Был изменен уровень шума со значения 0.2 на 0.0

Листинг 2 – Код функции question2()

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.0
    return answerDiscount, answerNoise
```

На рисунке 3 представлен результат автооценивания измененного параметра.



```
(piton36) C:\Users\Skory\piton36\МИСИИ_лаб_5>python autograder.py -q q2
Starting on 1-20 at 21:36:39

Question q2
=====

*** PASS: test_cases\q2\1-bridge-grid.test

### Question q2: 1/1 ###

Finished at 21:36:39

Provisional grades
=====
Question q2: 1/1
-----
Total: 1/1
```

Рисунок 3 – Результат тестирования кода для задания 2

Далее была введена команда: **`python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.0`**

Результат представлен на рисунке 4.

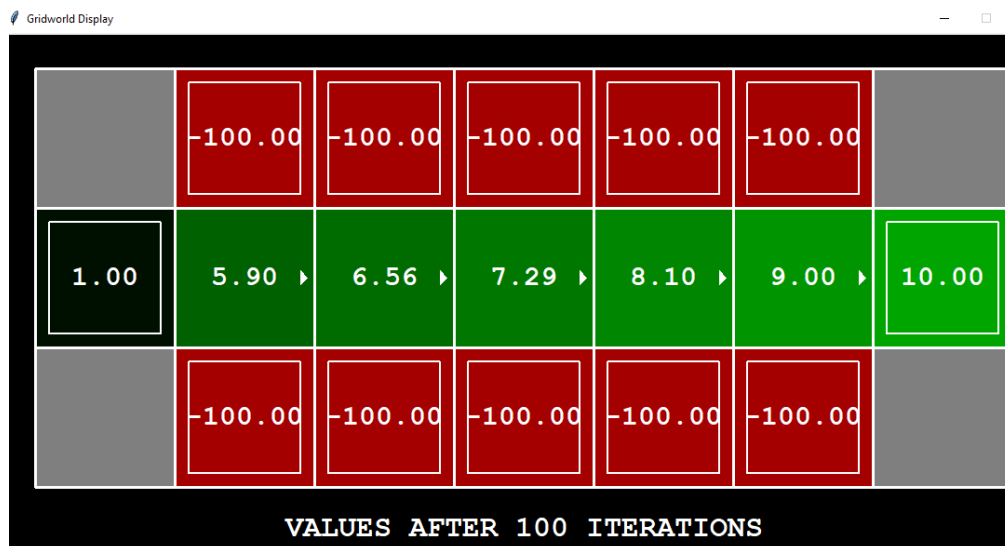


Рисунок 4 – Результат выполнения команды с параметром нулевого шума

Задание 3 (5 баллов). Реализация политик

Рассмотрим схему среды DiscountGrid, изображенную на рисунке 5.5. Эта среда имеет два терминальных состояния с положительной наградой (в средней строке): закрытый выход с наградой +1 и дальний выход с наградой +10. Нижняя строка схемы состоит из конечных состояний с отрицательными наградами -10 (показаны красным). Начальное состояние - желтый квадрат. Различают два типа путей: (1) пути, которые проходят по границе «обрыва» около нижнего ряда схемы: эти пути короче, но характеризуются большими отрицательными наградами (они обозначены красной стрелкой на рисунке 5.5); (2) пути, которые «избегают обрыва» и проходят по верхнему ряду схемы. Эти пути длиннее, но они с меньшей вероятностью принесут отрицательные результаты.

В этом задании необходимо подобрать значения коэффициента дисконтирования, уровня шума и текущей награды для DiscountGrid, чтобы сформировать оптимальные политики нескольких различных типов. Ваш выбор значений для указанных параметров должен обладать свойством, которое заключается в том, что если бы ваш агент следовал своей оптимальной политике, не подвергаясь никакому шуму, то он демонстрировал бы требуемое поведение. Если определенное поведение не достигается ни для одной

настройки параметров, необходимо сообщить, что политика невозможна, вернув строку «НЕВОЗМОЖНО».

Ниже указаны оптимальные типы политик, которые вы должны попытаться реализовать:

- 1) агент предпочитает близкий выход (+1), риск движения вдоль обрыва (-10);
- 2) агент предпочитает близкий выход (+1), но избегает обрыва (-10);
- 3) агент предпочитает дальний выход (+10), но рискует движением вдоль обрыва (-10);
- 4) агент предпочитает дальний выход (+10), избегает обрыва (-10);
- 5) агент предпочитает избегать оба выхода и обрыва (так что эпизод никогда не должен заканчиваться).

Внесите необходимые значения параметров в функции от question3a() до question3e() в файле analysis.py. Эти функции возвращают кортеж из трех элементов (дисконт, шум, награда).

Чтобы проверить свой выбор параметров, запустите автооценщик:

python autograder.py -q q3

Примечание. Вы можете проверить свои политики в графическом интерфейсе: **python gridworld.py -a value -i 100 -g DiscountGrid**.

Примечание. На некоторых машинах стрелка может не отображаться. В этом случае нажмите кнопку на клавиатуре, чтобы переключиться на дисплей qValue, и мысленно вычислите политику, взяв argmax от qValue для каждого состояния.

Оценивание: автооценщик проверяет, возвращается ли желаемая политика в каждом случае.

Реализация

В указанных в задании функциях были изменены параметры таким образом, чтобы все тесты были пройдены. В листинге 3 представлен полный код всех функций файла analysis.py.

Листинг 3 – Код функций от question3a() до question3e()

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.0
    return answerDiscount, answerNoise

def question3a():
    answerDiscount = 0.4
    answerNoise = 0.0
    answerLivingReward = -2.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3b():
    answerDiscount = 0.5
    answerNoise = 0.3
    answerLivingReward = -1.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3c():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = -2.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3d():
    answerDiscount = 0.9
    answerNoise = 0.3
    answerLivingReward = -1.0

    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3e():
    answerDiscount = 1.0
    answerNoise = 0.0
    answerLivingReward = 2.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

Код был протестирован с помощью автооценивания. Результат представлен на рисунке 5.

```
(python36) C:\Users\Skory\python36\МИСИИ_лаб_5>python autograder.py -q q3
Starting on 1-20 at 21:57:02

Question q3
=====

*** PASS: test_cases\q3\1-question-3.1.test
*** PASS: test_cases\q3\2-question-3.2.test
*** PASS: test_cases\q3\3-question-3.3.test
*** PASS: test_cases\q3\4-question-3.4.test
*** PASS: test_cases\q3\5-question-3.5.test

### Question q3: 5/5 ###

Finished at 21:57:03

Provisional grades
=====
Question q3: 5/5
-----
Total: 5/5
```

Рисунок 5 – Результаты тестирования для задания 3

Задание 4 (1 балл). Асинхронная итерация по значениям

Реализуйте агента, осуществляющего итерацию по значениям в классе `AsynchronousValueIterationAgent`, который частично определён в `valueIterationAgents.py`. При вызове конструктора класса `AsynchronousValueIterationAgent` на вход передается MDP и выполняется циклическая итерация по значениям для заданного количества итераций. Обратите внимание, что весь код итерации по значениям должен размещаться в конструкторе класса (метод `__init__`).

Причина, по которой этот класс называется `AsynchronousValueIterationAgent`, заключается в том, что на каждой итерации здесь обновляется только одно состояние, в отличие от выполнения пакетного обновления, когда обновляются все состояния. На первой итерации в классе `AsynchronousValueIterationAgent`, должно обновляться только значение первого состояния из списка состояний. На второй итерации обновляется только значение второго состояния и т.д. Процесс продолжается, пока не обновятся значения каждого состояния по одному разу, а затем происходит

возврат опять к первому состоянию. Если состояние, выбранное для обновления, является терминальным, на этой итерации ничего не происходит.

Класс `AsynchronousValueIterationAgent` является наследником от класса `ValueIterationAgent` из задания 1, поэтому единственный метод, который требуется изменить, — это `runValueIteration`. Поскольку конструктор суперкласса вызывает метод `runValueIteration`, достаточно его переопределения в классе `AsynchronousValueIterationAgent`, чтобы изменить поведение агента.

Примечание. Обязательно обработайте случай, когда состояние не имеет доступных действий в MDP (подумайте, что это означает для будущих вознаграждений).

Чтобы протестировать реализацию, запустите автооценщик. Время выполнения должно быть меньше секунды. Если это займет намного больше времени, то это позже приведет к проблемам при выполнении других заданий лабораторной работы, поэтому сделайте реализацию более эффективной сейчас.

`python autograder.py -q q4`

Следующая команда помещает `AsynchronousValueIterationAgent` в `Gridworld`, вычисляет политику и выполняет ее 10 раз.

`python gridworld.py -a asynchvalue -i 1000 -k 10`

Нажмите клавишу, чтобы просмотреть значения, Q-значения и ход моделирования. Можно обнаружить, что значение ценности начального состояния ($V(\text{start})$), которое можно посмотреть вне графического интерфейса) и эмпирическое результирующее среднее вознаграждение (напечатанное после 10 раундов выполнения) довольно близки.

Оценивание: агент итерации по значениям будет оцениваться с использованием новой схемы клеточного мира. Будут проверены значения ценностей состояний, Q-значения и политики после фиксированного количества итераций и при достижении сходимости (например, после 1000

итераций).

Реализация

Был написан код функции `runValueIteration`, объявленной в классе `AsynchronousValueIterationAgent(ValueIterationAgent)` в файле `valueIterationAgents.py`. Код представлен в листинге 4.

Листинг 4 – Код функции `runValueIteration`

```
def runValueIteration(self):
    """ ВСТАВЬТЕ ВАШ КОД СЮДА """
    states = self.mdp.getStates()
    for i in range(self.iterations):
        v = self.values.copy()
        s0 = states[i % len(states)]
        if self.mdp.isTerminal(s0):
            continue
        v[s0] = -float("inf")
        for a in self.mdp.getPossibleActions(s0):
            val = 0
            for s, p in self.mdp.getTransitionStatesAndProbs(s0, a):
                val += p * (self.mdp.getReward(s0, a, s) +
self.discount * self.values[s])
            v[s0] = max(v[s0], val)
        self.values = v.copy()
```

Написанная функция была протестирована с помощью автооценивания, результат представлен на рисунке 6.

Была введена команда: **`python gridworld.py -a asynchvalue -i 1000 -k 10`**

Результат выполнения команды представлен на рисунках 7-9.

```

C:\Windows\system32\cmd.exe
Starting on 1-20 at 22:40:10

Question q4
=====

*** PASS: test_cases\q4\1-tinygrid.test
*** PASS: test_cases\q4\2-tinygrid-noisy.test
*** PASS: test_cases\q4\3-bridge.test
*** PASS: test_cases\q4\4-discountgrid.test

### Question q4: 1/1 ###

Finished at 22:40:10

Provisional grades
=====
Question q4: 1/1
-----
Total: 1/1

```

Рисунок 6 – Результат тестирования кода для задания 4



Рисунок 7 – Значения после 1000 итераций

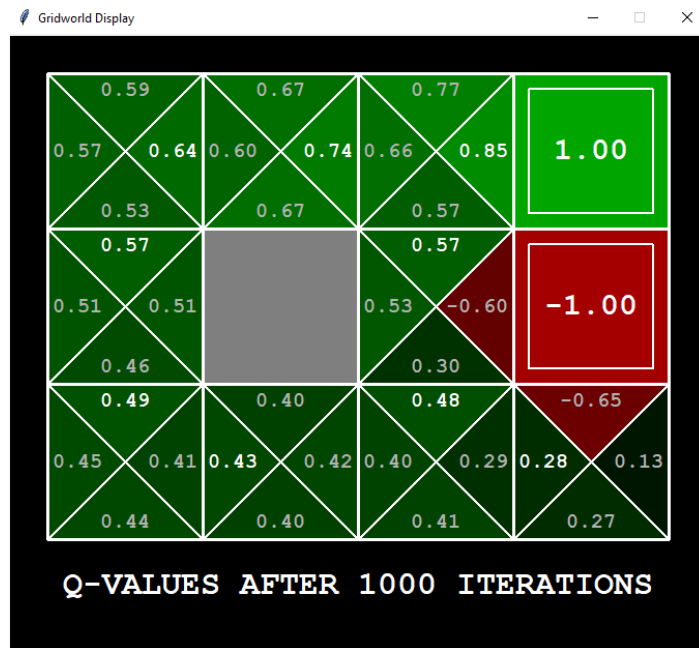


Рисунок 8 – Q-значения после 1000 итераций

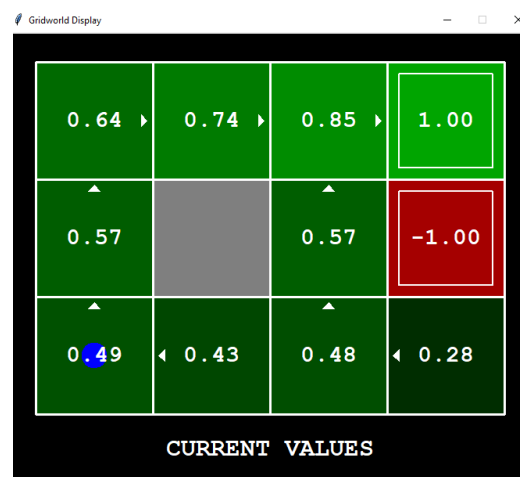


Рисунок 9 – Текущие значения

Задание 5 (3 балла). Итерации по приоритетным значениям

Реализуйте класс `PrioritizedSweepingValueIterationAgent`, который частично определен в `valueIterationAgents.py`. Обратите внимание, что этот класс является производным от `AsynchronousValueIterationAgent`, поэтому единственный метод, который необходимо изменить, — это `runValueIteration`, который фактически выполняет итерации по значениям.

В этом задании необходимо реализовать упрощенную версию стандартного алгоритма итераций по приоритетным значениям, который описан в статье. Будем использовать адаптированную версию этого алгоритма.

Во-первых, определим предшественниками состояния s все состояния, которые имеют ненулевую вероятность достижения s путем выполнения некоторого действия a . Кроме того, введем параметр θ , который будет представлять некоторый порог приращения функции ценности (устойчивость к ошибкам) при принятии решения об обновлении значения состояния. Сформулируем алгоритм, который вы должны реализовать:

1. Вычислите предшественников всех состояний;
2. Инициализируйте пустую приоритетную очередь;
3. Для каждого нетерминального состояния s выполните: (примечание: чтобы автооцениватель работал корректно необходимо перебирать состояния в порядке, возвращаемом `self.mdp.getStates()`):

- 3.1. Найдите абсолютное значение разности между текущим значением ценности s в `self.values` и наивысшим значением q -ценности для всех возможных действий из s ; назовите это значение именем `diff`. НЕ обновляйте `self.values[s]` на этом этапе;

- 3.2. Поместите s в очередь приоритетов с приоритетом `-diff` (обратите внимание, что это отрицательное значение). Используется отрицательное значение, потому что мы хотим отдавать приоритет обновлениям состояний, которые имеют более высокую ошибку;

4. Для `iteration` in `0, 1, 2, ..., self.iterations - 1` выполнить:
 - 4.1. Если приоритетная очередь пуста, завершите работу;
 - 4.2. Извлеките (метод `pop`) состояние s из очереди с приоритетами;
 - 4.3. Обновите значение ценности s (если это не конечное состояние) в `self.values`;

- 4.4. Для каждого p предшественника состояния s выполните:

- 4.4.1. Найдите абсолютное значение разности между текущим значением p в `self.values` и наивысшим значением q -ценности для всех возможных действий из p ; назовите это значение `diff`. НЕ обновляйте `self.values[p]` на этом этапе;

4.4.2. Если $\text{diff} > \text{theta}$, поместите p в очередь приоритетов с приоритетом $-\text{diff}$, если p отсутствует в очереди приоритетов с таким же или более низким приоритетом. Как и раньше, используется отрицательное значение приоритета, потому что приоритет отдается обновлению состояний, которые имеют более высокую ошибку.

Несколько важных замечаний к реализации алгоритма: когда определяются предшественники состояния, убедитесь, что они сохраняются во множестве, а не в списке, чтобы избежать дублирования; используйте `util.PriorityQueue` и метод `update` этого класса.

Чтобы протестировать вашу реализацию, запустите автооценщик. Время выполнения должно быть около 1 секунды. Если это займет намного больше времени, то это позже приведет к проблемам при выполнении других заданий лабораторной работы, поэтому сделайте реализацию более эффективной сейчас **`python autograder.py -q q5`**

Вы можете запустить `PrioritizedSweepingValueIterationAgent` в `Gridworld`, используя следующую команду.

`python gridworld.py -a priosweepvalue -i 1000`

Оценивание: агент с итерациями приоритетным значениям будет оцениваться с использованием новой схемы клеточного мира. Будут проверены q -ценности и политики после фиксированного количества итераций и при достижении сходимости (например, после 1000 итераций).

Реализация

Был разработан код функции, представленный в листинге 5.

Листинг 5 – Полный код класса `PrioritizedSweepingValueIterationAgent`

```
class
PrioritizedSweepingValueIterationAgent(AsynchronousValueIterationAgent):
    """
    * Пожалуйста, прочтите learningAgents.py перед тем, как читать
    это. *
```


Агент `PrioritizedSweepingValueIterationAgent` принимает марковский процесс принятия решения (см. `Mdp.py`) при инициализации и выполняет итерации по значениям с разверткой приоритетных состояний при заданном числе итераций с использованием предоставленных параметров.

```

"""
def __init__(self, mdp, discount = 0.9, iterations = 100, theta = 1e-
5):
    """
    Ваш агент итерации по развертке приоритетных значений должен
    принимать на вход МДП при создании, выполнять заданное количество
    итераций,

    а затем действовать в соответствии с полученной политикой.
    """
    self.theta = theta
    ValueIterationAgent.__init__(self, mdp, discount, iterations)

def runValueIteration(self):

    """ ВСТАВЬТЕ ВАШ КОД СЮДА """
    pred = self.generatePredecessors()
    pQ = util.PriorityQueue()
    states = self.mdp.getStates()
    for s in states:
        if self.mdp.isTerminal(s):
            continue
        pQ.push(s, -self.getDiff(s))
    for i in range(self.iterations):
        if pQ.isEmpty():
            break
        v = self.values.copy()
        s0 = pQ.pop()
        v[s0] = -float("inf")
        for a in self.mdp.getPossibleActions(s0):
            val = 0
            for s, p in self.mdp.getTransitionStatesAndProbs(s0, a):
                val += p * (self.mdp.getReward(s0, a, s) +
self.discount * self.values[s])
            v[s0] = max(v[s0], val)
        self.values = v.copy()
        for p in pred[s0]:

```

```

        diff = self.getDiff(p)
        if diff > self.theta:
            pQ.update(p, -diff)

def getDiff(self, s):
    qV = -float('inf')
    for a in self.mdp.getPossibleActions(s):
        qV = max(qV, self.getQValue(s, a))
    return abs(qV - self.values[s])

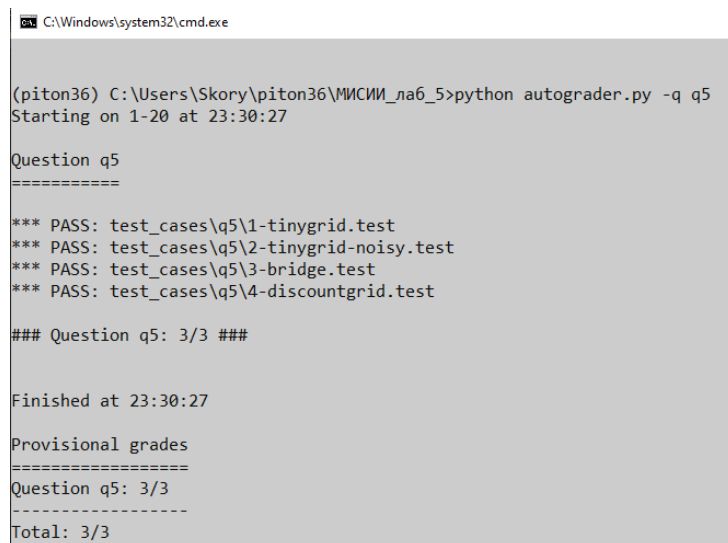
def generatePredecessors(self):
    pred = {}
    for s in self.mdp.getStates():
        for a in self.mdp.getPossibleActions(s):
            for next, p in self.mdp.getTransitionStatesAndProbs(s,
a):

                if p > 0:
                    if next not in pred:
                        pred[next] = {s}
                    else:
                        pred[next].add(s)

    return pred

```

Разработанный код был протестирован с помощью автооценивания, результаты – на рисунке 10.



```

C:\Windows\system32\cmd.exe

(piton36) C:\Users\Skory\piton36\МИСМИ_лаб_5>python autograder.py -q q5
Starting on 1-20 at 23:30:27

Question q5
=====

*** PASS: test_cases\q5\1-tinygrid.test
*** PASS: test_cases\q5\2-tinygrid-noisy.test
*** PASS: test_cases\q5\3-bridge.test
*** PASS: test_cases\q5\4-discountgrid.test

### Question q5: 3/3 ###

Finished at 23:30:27

Provisional grades
=====
Question q5: 3/3
-----
Total: 3/3

```

Рисунок 10 – Результат тестирования кода для задания 5

Задание 6 (4 балла). Q-обучение

Обратите внимание, что ваш агент итераций по значениям фактически не обучается на собственном опыте. Скорее, он обдумывает свою модель MDP, чтобы сформировать полную политику, прежде чем начнет взаимодействовать с реальной средой. Когда он действительно взаимодействует со средой, он просто следует предварительно вычисленной политике. Это различие может быть незаметным в моделируемой среде, такой как Gridworld, но очень важно в реальном мире, когда полное описание MDP отсутствует.

В этом задании необходимо реализовать агента с q-обучением, который мало занимается конструированием планов, но вместо этого учится методом проб и ошибок, взаимодействуя со средой с помощью метода `update(state, action, nextState, reward)`. Шаблон Q-обучения приведен в классе `QLearningAgent` в файле `qlearningAgents.py`, обращаться к нему можно из командной строки с параметрами `'-a q'`. Для этого задания необходимо реализовать методы `update`, `computeValueFromQValues`, `getQValue` и `computeActionFromQValues`.

Примечание. Для метода `computeActionFromQValues`, который возвращает лучшее действие в состоянии, при наличии нескольких действий с одинаковой q-ценностью, необходимо выполнить случайный выбор с помощью функции `random.choice()`. В некоторых состояниях действия, которые агент ранее не встречал, могут иметь значение q-ценности, в частности равное нулю, и если все действия, которые ваш агент встречал раньше, имеют отрицательное значение q-ценности, то действие, которое не встречалось может быть оптимальным.

Важно: убедитесь, что в функциях `computeValueFromQValues` и `computeActionFromQValues` вы получаете доступ к значениям q-ценности, вызывая `getQValue`. Эта абстракция будет полезна для задания 10, когда вы переопределите `getQValue` для использования признаков пар состояние-действие.

При использовании правила q-обновления, вы можете наблюдать за тем, как агент обучается, используя клавиатуру:

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

Напомним, что параметр `-k` контролирует количество эпизодов, которые агент использует для обучения. Посмотрите, как агент узнает о состоянии, в котором он только что был, а не о том, в которое он переходит, и «оставляет обучение на своем пути».

Подсказка: чтобы упростить отладку, вы можете отключить шум с помощью параметра `--noise 0.0` (хотя это делает q-обучение менее интересным). Если вы вручную направите `Rastan` на север, а затем на восток по оптимальному пути для четырех эпизодов, вы должны увидеть значения q-ценностей, указанные на рисунке 5.6.

Оценивание: проверяется, обучается ли агент тем же значениям q-ценности и политике, что и наша эталонная реализация на одном и том же наборе примеров.

Чтобы оценить вашу реализацию, запустите автооцениватель:

`python autograder.py -q q6`

Реализация

Были реализованы указанные в задании методы класса. В листинге 6 отображена реализация функций.

Листинг 6 – Методы `update`, `computeValueFromQValues`, `getQValue` и `computeActionFromQValues`.

```
def getQValue(self, state, action):  
  
    return self.QValue[(state, action)]  
  
def computeValueFromQValues(self, state):  
  
    best = float("-inf")  
  
    actions = self.getLegalActions(state)
```

```

        for action in actions:
            if best < self.getQValue(state, action):
                best = self.getQValue(state, action)

        if best != float("-inf"):
            return best
        else:
            return 0.0

def computeActionFromQValues(self, state):

    best = float("-inf")
    act = []

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

    for action in actions:
        if best < self.getQValue(state, action):
            best = self.getQValue(state, action)
            act = [action]
        elif best == self.getQValue(state, action):
            act.append(action)

    return random.choice(act)

def update(self, state, action, nextState, reward):

    self.QValue[(state, action)] = (1-
self.alpha)*self.QValue[(state, action)] + self.alpha*(reward +
self.discount*self.computeValueFromQValues(nextState))

```

Были проведены тесты. Результаты представлены на рисунке 11.

```

(piton36) C:\Users\Skory\piton36\МИСИИ_лаб_5>python autograder.py -q q6
Starting on 1-21 at 0:10:40

Question q6
=====

*** PASS: test_cases\q6\1-tinygrid.test
*** PASS: test_cases\q6\2-tinygrid-noisy.test
*** PASS: test_cases\q6\3-bridge.test
*** PASS: test_cases\q6\4-discountgrid.test

### Question q6: 4/4 ###

Finished at 0:10:40

Provisional grades
=====
Question q6: 4/4
-----
Total: 4/4

```

Рисунок 11 – Результаты тестирования для задания 6

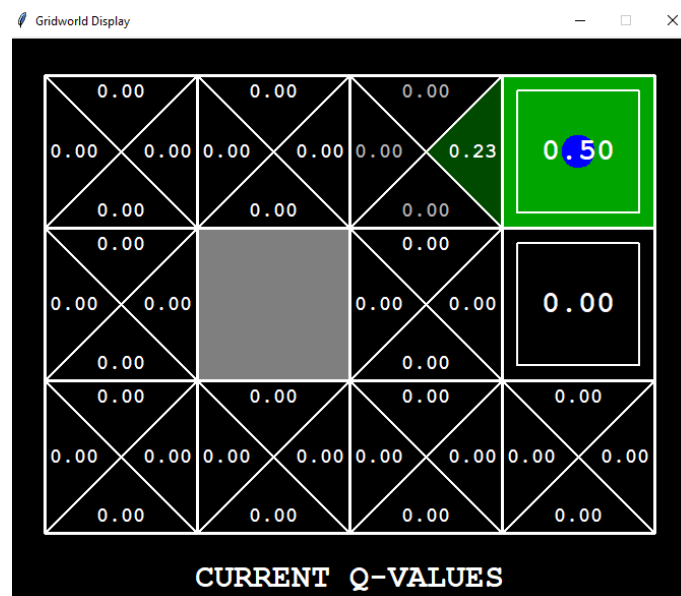


Рисунок 12 – Результаты q-обучения

Задание 7 (2 балла). Эпсилон-жадная стратегия

Завершите реализацию агента с q-обучением, дописав эпсилон-жадную стратегию выбора действий в методе `getAction`. Метод должен обеспечивать выбор случайного действия с вероятностью эпсилон и в противном случае выбирать действие с лучшим значением q-ценности. Обратите внимание, что выбор случайного действия может привести к выбору наилучшего действия, то есть вам следует выбирать не случайное неоптимальное действие, а любое случайное допустимое действие.

Вы можете выбрать действие из списка случайным образом, вызвав функцию `random.choice`. Вы можете смоделировать двоичную переменную с

вероятностью успеха p , используя вызов `util.flipCoin(p)`, который возвращает `True` с вероятностью p и `False` с вероятностью $1-p$.

После реализации метода `getAction` обратите внимание на следующее поведение агента в `gridworld` (с `epsilon = 0.3`).

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

Ваши окончательные значения q -ценностей должны соответствовать значениям, получаемым при итерации по значениям, особенно на проторенных путях.

Однако среднее вознаграждение будет ниже, чем предсказывают q -ценности из-за случайных действий и начальной фазы обучения.

Вы также можете наблюдать поведение агента при разных значениях эпсилон (параметр `-e`). Соответствует ли такое поведение агента вашим ожиданиям?

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.1
```

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

Чтобы протестировать вашу реализацию, запустите автооценщик:

```
python autograder.py -q q7
```

Теперь без дополнительного кодирования вы сможете запустить q -обучение для робота `crawler`:

```
python crawler.py
```

Если вызов не работает, то вероятно, вы написали слишком специфичный код для задачи `GridWorld`, и вам следует сделать его более общим для всех `MDP`.

При корректной работе на экране появится ползущий робот, управляемый из класса `QLearningAgent`. Поиграйте с различными параметрами обучения, чтобы увидеть, как они влияют на действия агента. Обратите внимание, что параметр `step delay` (задержка шага) является параметром моделирования, тогда как скорость обучения и эпсилон являются параметрами алгоритма обучения, а коэффициент дисконтирования является

свойством среды.

Реализация

Требовалось завершить реализацию агента с q-обучением, дописав эпсилон-жадную стратегию выбора действий в методе `getAction`. Метод должен обеспечивать выбор случайного действия с вероятностью эпсилон и в противном случае выбирать действие с лучшим значением q-ценности. Написанный метод представлен в листинге 7.

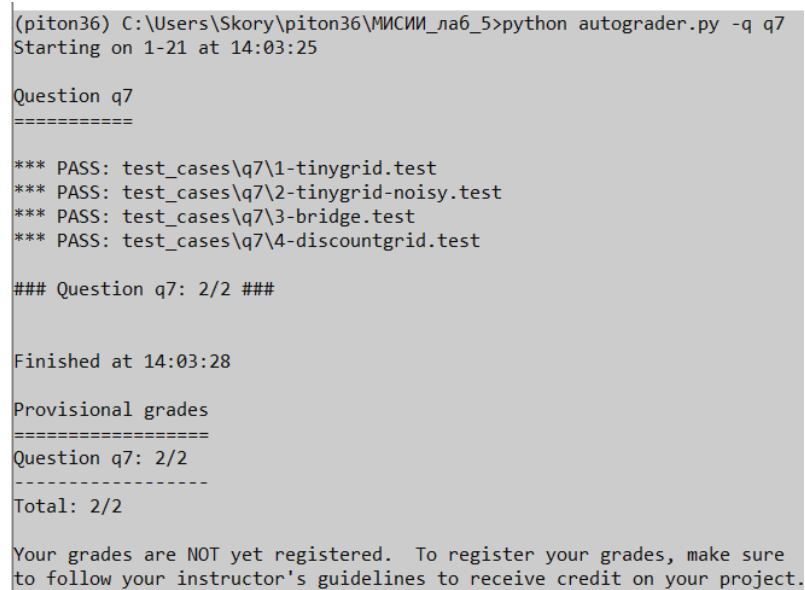
Листинг 7 – Реализация функции `getAction`

```
def getAction(self, state):

    legalActions = self.getLegalActions(state)

    if util.flipCoin(self.epsilon):
        return random.choice(legalActions)
    else:
        return self.computeActionFromQValues(state)
```

Написанная функция была подвергнута тестированию, результат на рисунке 13.



```
(piton36) C:\Users\Skory\piton36\МИСМИ_лаб_5>python autograder.py -q q7
Starting on 1-21 at 14:03:25

Question q7
=====

*** PASS: test_cases\q7\1-tinygrid.test
*** PASS: test_cases\q7\2-tinygrid-noisy.test
*** PASS: test_cases\q7\3-bridge.test
*** PASS: test_cases\q7\4-discountgrid.test

### Question q7: 2/2 ###

Finished at 14:03:28

Provisional grades
=====
Question q7: 2/2
-----
Total: 2/2

Your grades are NOT yet registered. To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Рисунок 13 – Результат тестирования кода для задания 7

Была введена команда: **python gridworld.py -a q -k 100**

Результат представлен на рисунке 14.

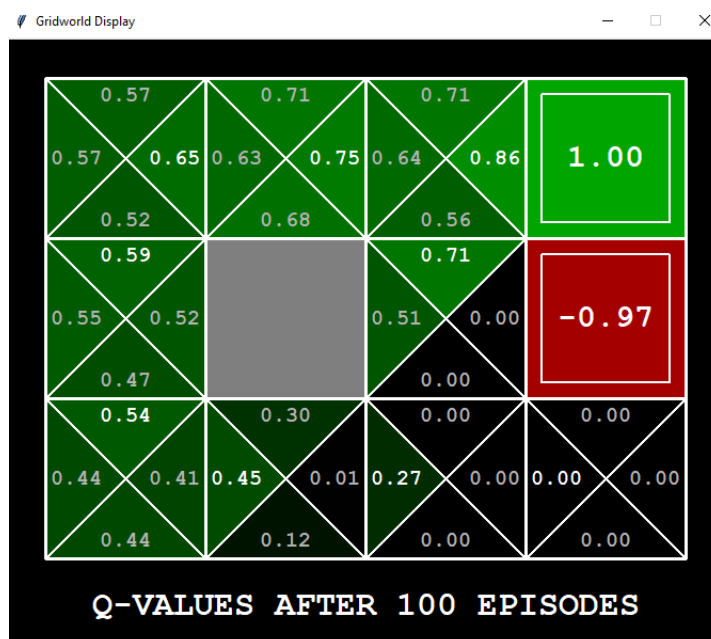


Рисунок 14 – Окончательные значения q-ценностей

Была введена команда **python gridworld.py -a q -k 100 --noise 0.0 -e 0.9**

Результат представлен на рисунке 15.

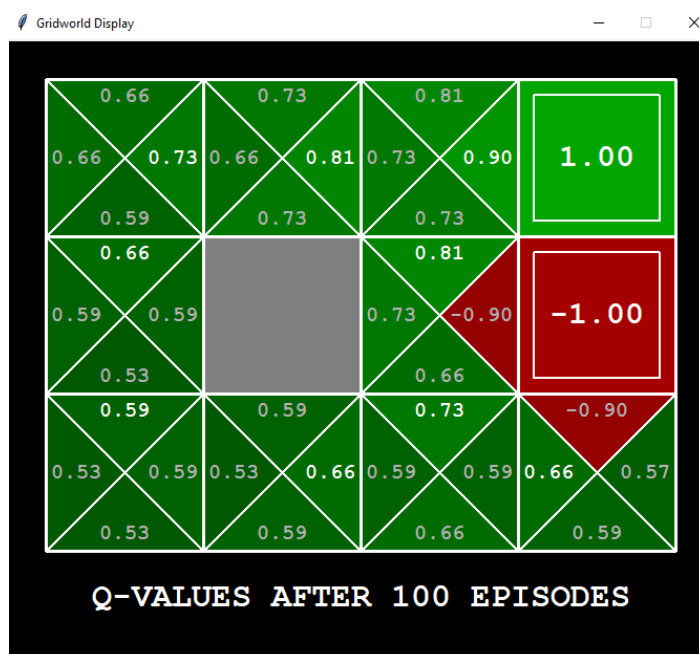


Рисунок 15 – Поведение агента при значении эпсилон 0.9

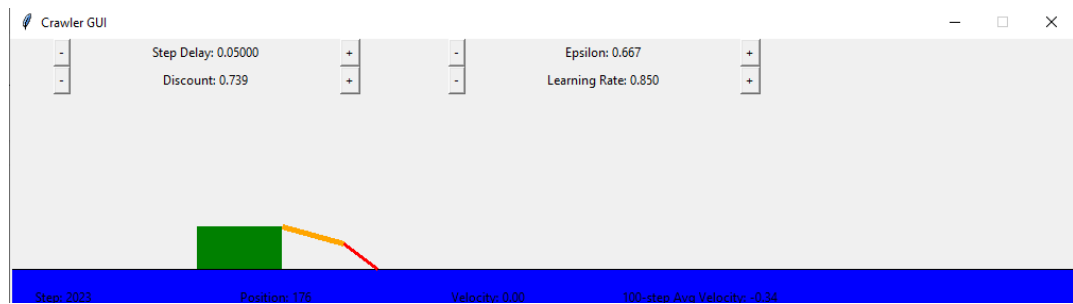


Рисунок 16 – Q-обучение для робота crawler

Задание 8 (1 балл). Переход по мосту

Обучите агента с q-обучением при полностью случайном выборе действий со скоростью обучения, заданной по умолчанию, отсутствии шума в среде BridgeGrid. Проведите обучение на 50 эпизодах и наблюдайте, найдет ли агент оптимальную политику:

`python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1`

Теперь попробуйте тот же эксперимент с $\epsilon = 0.0$. Есть ли такие значения эпсилон и скорости обучения, для которых высока вероятность (более 99%) того, что агент обучится оптимальной политике после 50 итераций? Ответ разместите в функции `question8()` в файле `analysis.py`. Эта функция должна возвращать либо двухэлементный кортеж (`epsilon`, `learning rate`), либо строку 'NOT POSSIBLE', если таких значений нет. Epsilon управляется параметром `-e`, скорость обучения параметром `-l`.

Примечание. Ваш ответ не должен зависеть от точного механизма разрешения конфликтов, используемого для выбора действий. Это означает, что ваш ответ должен быть правильным, даже если, например, мы повернули бы всю схему моста на 90 градусов.

Чтобы оценить свой ответ, запустите автооценщик:

`python autograder.py -q q8`

Реализация

В листинге 8 представлено решение задания.

Листинг 8 – Функция `question8()` в файле `analysis`

```
def question8():
```

```

answerEpsilon = None
answerLearningRate = None
return 'NOT POSSIBLE'

```

На рисунке 17 представлены результаты тестирования написанной функции.

```

(piton36) C:\Users\Skory\piton36\МИСМИ_лаб_5>python autograder.py -q q8
Starting on 1-21 at 14:56:31

Question q8
=====

*** PASS: test_cases\q8\grade-agent.test

### Question q8: 1/1 ###

Finished at 14:56:31

Provisional grades
=====
Question q8: 1/1
-----
Total: 1/1

Your grades are NOT yet registered. To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

```

Рисунок 17 – Результаты тестирования кода для задания 8

На рисунке 18 представлен результат работы команды **python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1**.



Рисунок 18 – Обучение на 50-ти эпизодах при эпсилоне 1

Задание 9 (1 балл). Q-обучение и Пакман

Пора поиграть в Пакман! Игра будет проводиться в два этапа. На первом этапе обучения Пакман начнет обучаться значениям ценности позиций и действиям. Поскольку получение точных значений q-ценностей даже для крошечных полей игры занимает очень много времени, обучение Пакмана по умолчанию выполняется без отображения графического интерфейса (или консоли). После завершения обучения Pacman перейдет в режим тестирования. При тестировании для параметров `self.epsilon` и `self.alpha` будут установлено значение 0, что фактически остановит q-обучение и отключит режим исследования, чтобы позволить Пакману использовать сформированную в ходе обучения политику. По умолчанию тестовая игра отображается в графическом интерфейсе. Без каких-либо изменений кода вы сможете запустить q-обучение Пакмана для маленького поля игры следующим образом:

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

Обратите внимание, что `PacmanQAgent` уже определен для вас в терминах уже написанного агента `QLearningAgent`. `PacmanQAgent` отличается только тем, что он имеет параметры обучения по умолчанию, которые более эффективны для игры Пакман (`epsilon=0.05`, `alpha=0.2`, `gamma=0.8`). Вы получите максимальную оценку за выполнение этого задания, если приведенная выше команда будет выполняться без исключений и агент выигрывает не менее, чем в 80% случаев. Автооценщик проведет 100 тестовых игр после 2000 тренировочных игр.

Подсказка: если агент `QLearningAgent` успешно работает с `gridworld.py` и `crawler.py`, но при этом не обучается хорошей политике для игры Пакман на поле `smallGrid`, то это может быть связано с тем, что методы `getAction` и/или `computeActionFromQValues` в некоторых случаях не учитывают должным образом ненаблюдаемые действия. В частности, поскольку ненаблюдаемые действия по определению имеют нулевое значение q-ценности, а все действия, которые были исследованы, имели отрицательные значения q-ценности, то не

наблюдаемое действие может быть оптимальным. Остерегайтесь применения функции `argmax` класса `util.Counter`.

Чтобы оценить задание 9, выполните:

`python autograder.py -q q9`

Примечание. Если вы хотите поэкспериментировать с параметрами обучения, вы можете использовать параметр `-a`, например `-a epsilon=0.1, alpha=0.3, gamma=0.7`. Затем эти значения будут доступны как `self.epsilon`, `self.gamma` и `self.alpha` внутри агента.

Примечание. Хотя всего будет сыграно 2010 игр, первые 2000 игр не будут отображаться из-за опции `-x 2000`, которая обозначает, что первые 2000 обучающих игр не отображаются. Таким образом, вы увидите, что `Pacman` играет только в последние 10 из этих игр. Количество обучающих игр также передается вашему агенту в качестве опции `numTraining`.

Примечание. Если вы хотите посмотреть 10 тренировочных игр, чтобы узнать, что происходит, используйте команду:

`python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10`

Во время обучения результаты будут отображаться после каждых 100 игр с демонстрацией статистики. Эпсилон-жадная стратегия дает положительные результаты во время обучения, поэтому `Pacman` играет плохо даже после того, как усвоит хорошую политику: это потому, что он иногда продолжает делать случайный исследовательский ход в сторону призрака. Для сравнения: должно пройти от 1000 до 1400 игр, прежде чем `Pacman` получит положительное вознаграждение за сегмент из 100 эпизодов, что свидетельствует о том, что он начал больше выигрывать, чем проигрывать. К концу обучения вознаграждение должно оставаться положительным и быть достаточно высоким (от 100 до 350).

После того, как `Pacman` закончил обучение, он должен очень надежно выигрывать в тестовых играх (по крайней мере, в 90% случаев), поскольку теперь он использует обученную политику.

Однако вы обнаружите, что обучение того же агента, казалось бы, на простой среде `mediumGrid` не работает. В нашей реализации среднее вознаграждение за обучение Пакмана остается отрицательным на протяжении всего обучения. Во время теста он играет плохо, вероятно, проигрывая все свои тестовые партии.

Тренировки тоже займут много времени, несмотря на свою неэффективность.

Пакман не может победить на больших игровых полях, потому что каждая конфигурация имеет отдельные состояния с отдельными значениями q -ценности.

У Пакмана нет возможности обобщить случаи столкновения с призраком и понять, что это плохо для всех позиций. Очевидно, такой вариант q -обучения не масштабирует большое число состояний.

Реализация

Было запущено q -обучение Пакмана для маленького поля игры следующим образом:

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

Результаты выполнения команды представлены на рисунке 19.

```
Completed 1900 out of 2000 training episodes
Average Rewards over all training: -88.06
Average Rewards for last 100 episodes: 287.62
Episode took 2.67 seconds
Reinforcement Learning Status:
Completed 2000 out of 2000 training episodes
Average Rewards over all training: -67.24
Average Rewards for last 100 episodes: 328.38
Episode took 2.12 seconds
Training Done (turning off epsilon and alpha)
-----
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Average Score: 498.6
Scores:      503.0, 499.0, 495.0, 499.0, 503.0, 499.0, 495.0, 499.0, 495.0, 499.0
Win Rate:    10/10 (1.00)
Record:      Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Рисунок 19 – Результаты игры Пакмана после q -обучения

Была введена команда тестирования автооцениванием. Результат – на рисунке 20.

[illegible]

Рисунок 20 – Результаты тестирования задания 9

Задание 10 (3 балла). Q-обучение с аппроксимацией

Реализуйте q-обучение с аппроксимацией, которое обеспечивает обучение весов признаков состояний. Дополните описание класса `ApproximateQAgent` в `qlearningAgents.py`, который является подклассом `РасmanQAgent`. Q-обучение с аппроксимацией предполагает существование признаковой функции $f(s, a)$ от пары состояние-действие, которая возвращает вектор $f_1(s, a) \dots f_i(s, a) \dots f_n(s, a)$ из значений признаков. Вам предоставляются для этого возможности модуля `featureExtractors.py`. Векторы признаков - это объекты `util.Counter` (подобны словарю), содержащие ненулевые пары признаков и значений; все пропущенные признаки будут иметь нулевые значения.

По умолчанию ApproximateQAgent использует IdentityExtractor, который назначает один признак каждой паре состояние-действие. С этой функцией извлечения признаков агент, осуществляющий q-обучение с аппроксимацией, будет работать идентично PacmanQAgent. Вы можете проверить это с помощью следующей команды:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

Важно: `ApproximateQAgent` является подклассом `QLearningAgent` и поэтому имеет с ним несколько общих методов, например, `getAction`. Убедитесь, что методы в `QLearningAgent` вызывают `getQValue` вместо прямого доступа к q-ценностям, чтобы при переопределении `getQValue` в вашем агенте использовались новые аппроксимированные q-ценности для определения действий.

Убедившись, что агент, основанный на аппроксимации состояний, правильно работает, запустите его с использованием `SimpleExtractor` для извлечения пользовательских признаков, который с легкостью научится побеждать:

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

Даже более сложные игровые поля не должны стать проблемой для `ApproximateQAgent` (предупреждение: обучение может занять несколько минут):

```
python pacman.py -p ApproximateQAgent -a  
extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic
```

Если не будет ошибок, то агент с q-обучением и аппроксимацией должен будет почти каждый раз выигрывать при использовании этих простых признаков, даже если у вас всего 50 обучающих игр.

Оценивание: будет проверяться, что ваш агент обучается тем же значениям q-ценностей и весам признаков, что и эталонная реализация. Чтобы оценить вашу реализацию, запустите автооцениватель:

```
python autograder.py -q q10
```

Реализация

В листинге 9 представлен полный код класса `ApproximateQAgent` в `qlearningAgents.py`.

Листинг 9 – Код класса `ApproximateQAgent`

```
class ApproximateQAgent(PacmanQAgent):
```



```

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):

    weights = self.getWeights()
    features = self.featExtractor.getFeatures(state, action)
    q = weights * features
    return q

def update(self, state, action, nextState, reward):

    weights = self.getWeights()
    features = self.featExtractor.getFeatures(state, action)

    nextActions = self.getLegalActions(nextState)
    maxNextQ = float("-inf")

    for act in nextActions:
        q = self.getQValue(nextState, act)
        maxNextQ = max(maxNextQ, q)

    if len(nextActions) == 0:
        maxNextQ = 0.0

    difference = reward + self.discount*maxNextQ -
self.getQValue(state, action)
    for feature in features:
        weights[feature] = weights[feature] + self.alpha * difference
    * features[feature]

def final(self, state):
    PacmanQAgent.final(self, state)

    if self.episodesSoFar == self.numTraining:

        pass

```

Было проведено тестирование разработанного кода. Результаты представлены на рисунке 21.

```
(python36) C:\Users\Skory\python36\МИСИИ_лаб_5>python autograder.py -q q10
Starting on 1-21 at 15:53:49

Question q10
=====

*** PASS: test_cases\q10\1-tinygrid.test
*** PASS: test_cases\q10\2-tinygrid-noisy.test
*** PASS: test_cases\q10\3-bridge.test
*** PASS: test_cases\q10\4-discountgrid.test
*** PASS: test_cases\q10\5-coord-extractor.test

### Question q10: 3/3 ###

Finished at 15:53:49

Provisional grades
=====
Question q10: 3/3
-----
Total: 3/3
```

Рисунок 21 – Тестирование кода для задания 10

Далее была введена команда: **python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid**. Результат представлен на рисунке 22.

```
C:\Windows\system32\cmd.exe
Completed 1900 out of 2000 training episodes
Average Rewards over all training: -67.96
Average Rewards for last 100 episodes: 257.20
Episode took 5.11 seconds
Reinforcement Learning Status:
Completed 2000 out of 2000 training episodes
Average Rewards over all training: -51.22
Average Rewards for last 100 episodes: 266.72
Episode took 4.65 seconds
Training Done (turning off epsilon and alpha)
-----
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 503
Average Score: 500.5
Scores: 495.0, 502.0, 495.0, 499.0, 502.0, 503.0, 502.0, 502.0, 502.0, 503.0
Win Rate: 10/10 (1.00)
Record: Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Рисунок 22 – Результаты игры агента, основанного на аппроксимации состояний

Была введена команда: **python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic**

Результат представлен на рисунке 23.

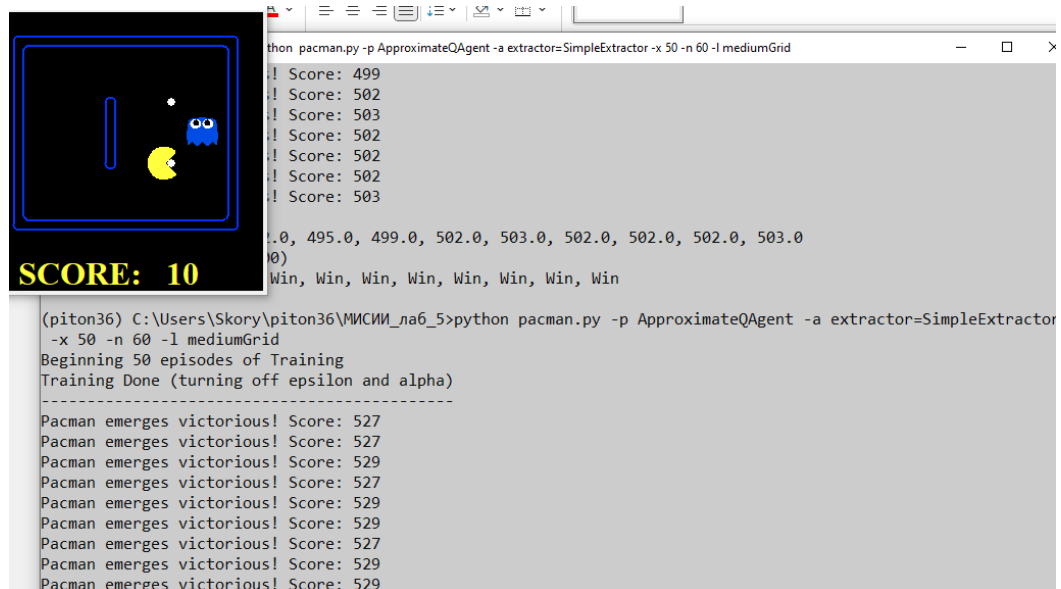


Рисунок 23 – Игра того же агента на более сложном игровом поле

В приложении А представлен полный код файлов `valueIterationAgents.py`, `qlearningAgents.py`, `analysis.py`.

4. Выводы

В ходе выполнения лабораторной работы было проведено исследование методов недетерминированного поиска решений задач, приобретены навыки программирования интеллектуальных агентов, функционирующих в недетерминированных средах, проведено исследование методов построения агентов, обучаемых на основе алгоритмов обучения с подкреплением.

5. Контрольные вопросы

5.1 Объясните, что понимают под недетерминированными задачами поиска?

Рассмотрим методы поиска в условиях, когда действие агента, выполненное в некотором состоянии, может приводить ко многим возможным

новым состояниям с определенной долей вероятности. Такие задачи поиска, в которых поведение агента характеризуется степенью неопределенности, относятся к недетерминированным задачам поиска. Они могут быть решены с помощью моделей, известных как марковские процессы принятия решений (Markov Decision Processes – MDP).

Недетерминированные задачи поиска — это класс задач, в которых присутствует неопределенность и неоднозначность в процессе поиска оптимального решения. В отличие от детерминированных задач поиска, где каждый шаг явно определен и не оставляет места для неопределенности, недетерминированные задачи поиска предоставляют несколько возможных вариантов действий на каждом шаге.

В недетерминированных задачах поиска возможны различные ситуации:

1. Несколько возможных действий: На каждом шаге поиска агент может иметь несколько возможных действий для выбора. Например, в задаче поиска пути в лабиринте, на каждом шаге агент может иметь несколько направлений, в которые он может двигаться.

2. Несколько возможных состояний: Вместо того, чтобы иметь одно конкретное следующее состояние после выполнения действия, агент может оказаться в нескольких возможных состояниях. Например, в задаче поиска пути в лабиринте с неизвестными перемещающимися препятствиями, следующее состояние агента может быть неопределенным из-за движения препятствий.

3. Несколько возможных исходов: Даже если агент выбирает одно действие и окажется в одном состоянии, результат выполнения действия может быть неоднозначным. Например, в игре с неопределенными перемещениями противников, результат атаки может быть случайным и неопределенным.

5.2. Объясните основные понятия Марковского процесса принятия решений.

5.3. Что понимается под функцией политики?

5.4. Что понимается под функцией полезности?

5.5. Какая задача называется эпизодической?

5.7. Запишите выражение для вычисления функции полезности для продолжающихся задач.

Для продолжавшихся задач, у которых нет завершающего состояния (в отличие от эпизодических задач) вводится понятие коэффициента дисконтирования γ . Определим функцию полезности (возврат) с коэффициентом дисконтирования в следующем виде:

$$U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \dots$$

Коэффициент γ определяет относительную важность будущих и немедленных наград. Его значение лежит в диапазоне от 0 до 1; $\gamma=0$ означает, что немедленные награды более важны, а $\gamma=1$ означает, что будущие награды важнее немедленных. Для модели бесконечного горизонта значение функции полезности определяется выражением ($\gamma < 1$):

$$U([r_0, \dots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1 - \gamma)$$

5.8. Как определяется функция ценности состояния?

Функция ценности состояния (state value function) или просто «функция ценности» указывает, насколько хорошо для агента пребывание в конкретном состоянии s при следовании политике π . Эта функция равна ожидаемой кумулятивной награде при следовании политике π в состоянии s :

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi \right]$$

где \mathbb{E} – символ математического ожидания.

5.9. Как определяется q-функция ценности состояния-действия?

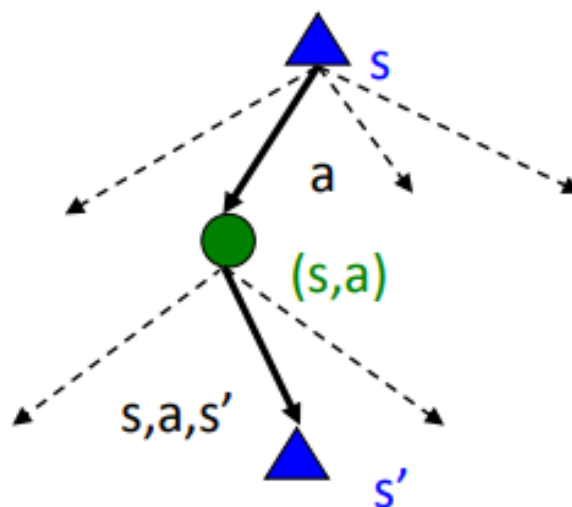
Q-функция ценности состояния-действия (s, a) равна ожидаемой кумулятивной награде при выборе действия a в состоянии s и при следовании политике π :

$$Q^{\pi}(s, a) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right] \quad (5.6)$$

Q -функция, в отличие от функции ценности состояния, скорее *определяет полезность действия в состоянии*, а не полезность самого состояния.

5.10. Объясните, как строится MDP дерево поиска?

MDP, также как поиск в пространстве состояний, можно представить в виде дерева поиска (рисунок 5.1). Неопределенность моделируется в этих деревьях с помощью q -состояний, также известных как узлы **состояния-действия** (s, a), по существу идентичных узлам жеребьевки в ExpeTimax; q -состояние представляется в виде действия a в состоянии s и обозначается как кортеж (s, a) ; **q -состояния** используют вероятности для моделирования неопределенности того, что агент перейдет в следующее состояние s' .



Построение MDP дерева поиска включает несколько шагов:

1. **Определение состояний:** Сначала необходимо определить состояния, в которых может находиться агент. Состояния представляют собой различные ситуации внутри MDP, которые могут влиять на принимаемые решения. Например, в игре в крестики-нолики состояниями могут быть различные расстановки фишек на игровом поле.

2. Определение действий: Для каждого состояния определяются возможные действия, которые агент может предпринять. Действия могут быть ограничены или неограничены в зависимости от контекста. Например, в игре в крестики-нолики действиями могут быть размещение фишки в определенную клетку игрового поля.

3. Определение вероятностей переходов: Для каждого состояния и действия определяются вероятности переходов в новые состояния. Эти вероятности указывают, с какой вероятностью агент окажется в каждом из возможных следующих состояний после выполнения действия. Например, в игре в крестики-нолики вероятности переходов могут зависеть от действий другого игрока или случайных факторов.

4. Определение вознаграждений: Для каждого состояния и действия определяются вознаграждения или штрафы, связанные с переходом в следующее состояние. Вознаграждения могут быть положительными или отрицательными и указывать на ценность или нежелательность перехода. Например, в игре в крестики-нолики положительное вознаграждение может быть назначено за выигрыш, а отрицательное - за проигрыш.

5. Построение дерева: После определения состояний, действий, вероятностей переходов и вознаграждений строится MDP дерево поиска. В этом дереве каждый узел представляет состояние, каждое ребро - действие, и ассоциированное с ребром значение - вероятность перехода и вознаграждение. Дерево строится в соответствии с возможными последовательностями действий и состояний в MDP.

5.11. Запишите и объясните уравнение Беллмана для функции ценности состояния.

Подставив (5.10) в (5.9), получим уравнение Беллмана

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \quad , \quad (5.11)$$

где $T(s, a, s')$ – вероятность перехода из (s, a) в s' , а $R(s, a, s') + \gamma V^*(s')$ – ценность действия a в состоянии s при переходе в s' . Уравнение Беллмана устанавливает рекуррентную связь между ценностью состояния s и ценностью следующего состояния s' при выполнении наилучшего действия a .

5.12. Сформулируйте и объясните алгоритм итерации по значениям ценности состояний.

Итерации по значениям $V(s)$ – это алгоритм динамического программирования, который обеспечивает итеративное вычисления ценности состояний $V(s)$, пока не выполнится условие сходимости:

$$\forall s, V_{k+1}(s) = V_k(s). \quad (5.12)$$

Алгоритм предусматривает следующие шаги:

1. Положить $V_0(s) = 0$: отсутствие действий на шаге 0 означает, что ожидаемая сумма наград равна нулю;
2. Для каждого из состояний повторять вычисление (обновлять) ценности состояния в соответствии с выражением, пока значения не сойдутся

$$\forall s \in S, V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \quad (5.13)$$

Временная сложность каждой итерации алгоритма соответствует $O(S^2A)$. Алгоритм обеспечивает схождение ценности каждого состояния к оптимальному значению. При этом следует отметить, что политика может сходиться долго до того, как сойдутся ценности состояний.

5.13. Сформулируйте и объясните метод извлечения политики.

Цель решения MDP – определение оптимальной политики. Предположим, что имеются оптимальные значения ценности состояний $V^*(s)$. Каким образом следует при этом действовать в каждом состоянии? Это можно определить, применив метод, который называется **извлечением политики** (policy extraction). Если агент находится в состоянии s , то следует выбрать действие a , обеспечивающее получение максимальной ожидаемой суммарной награды:

$$\pi^*(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \quad (5.14)$$

Не удивительно, что a является действием, которое приводит к q -состоянию с максимальным значением q -ценности, которое формально соответствует определению оптимальной политики:

$$\pi^*(s) = \arg \max_a Q^*(s, a) \quad (5.15)$$

Таким образом, *проще выбирать действия по q -ценностям, чем по ценностям состояний!*

5.14. Сформулируйте и объясните алгоритм итерации по политикам.

Итерации по значениям функций ценности могут быть очень медленными. Поэтому, когда требуется определить политику, можно использовать альтернативный подход, основанный на итерациях по политикам. В соответствии с этим подходом, предусматриваются следующие шаги поиска оптимальной политики:

1. Определить первоначальную политику π . Такая политика может быть произвольной;
2. Выполнить оценку текущей политики, используя метод **оценки политики** (policy evaluation). Для текущей политики π оценка политики означает вычисление $V^\pi(s)$ для всех состояний:

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V^\pi(s')] \quad (5.16)$$

Вычисление $V^\pi(s)$ можно выполнить *двумя способами*: либо решить систему линейных уравнений (5.16) относительно $V^\pi(s)$ (например, в Matlab); либо вычислить оценки состояний итерационно, используя пошаговые обновления:

$$V_0^\pi(s) = 0, \\ V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')] \quad (5.17)$$

Обозначить текущую политику как π_i . Временная сложность итерационной оценки политики соответствует $O(S^2)$ на 1 итерацию. Тем не менее, второй способ оценки политики значительно медленнее на практике.

3. После того как выполнена оценка текущей политики π_i , выполняется **улучшение политики** (policy improvement) на основе одношагового метода извлечения политики:

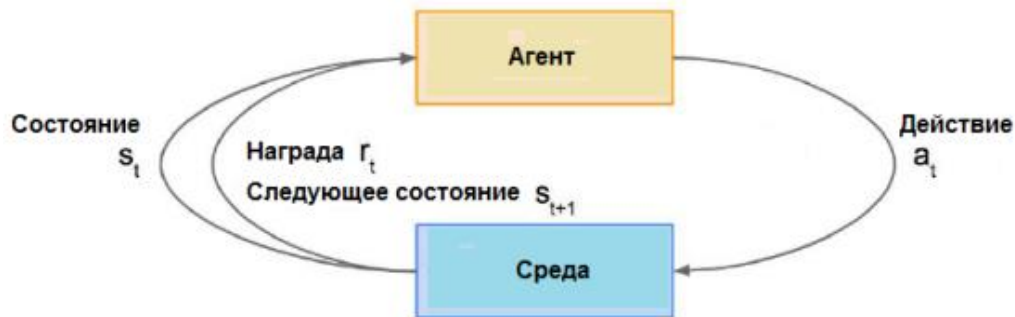
$$\pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^{\pi_i}(s')] \quad (5.18)$$

Если $\pi_{i+1} = \pi_i$, то алгоритм останавливают и полагают $\pi_{i+1} = \pi_i = \pi^*$, иначе переходят к п.2.

5.15. Объясните основные понятия обучения с подкреплением.

Обучение с подкреплением (RL, reinforcement learning) — область машинного обучения, в которой обучение осуществляется посредством

взаимодействия агента с окружающей средой (рисунок 5.2) [7, 8]. В среде RL вы не указываете агенту, что и как он должен делать, вместо этого **вы даете агенту награду за каждое выполненное действие**. Тогда агент начинает выполнять действия, которые максимизируют вознаграждение.



Предполагается, что среда описывается марковским процессом принятия решений (MDP). Но если в методах итерации по значениям и итерации по политикам нам были известны функции переходов T и вознаграждений R , то при обучении с подкреплением эти функции неизвестны.

При обучении с подкреплением агент предпринимает попытки **исследования (exploration)** среды, совершая действия и получая обратную связь в форме новых состояний и наград. Агент использует эту информацию для определения оптимальной политики в ходе процесса, называемого RL, прежде чем начнет **эксплуатировать (exploitation)** полученную политику.

Последовательность взаимодействия агента со средой (s , a , s' , r) на одном шаге называют **выборкой**. Коллекция выборок, которая приводит к терминальному состоянию, называется **эпизодом**.

Агент обычно выполняет много эпизодов в ходе исследования для того, чтобы собрать достаточно данных для обучения.

Модель является представлением среды с точки зрения агента. Различают два типа обучения с подкреплением: **основанное на модели и без модели**.

5.16. Что понимается под исследованиями и эксплуатацией при RL обучении?

Исследование (exploration): Исследование относится к стратегии, при которой агент исследует окружающую среду, чтобы получить новые знания и опыт. Во время исследования агент может предпринимать случайные или неопределенные действия, чтобы исследовать неизвестные или малоизученные области среды. Цель исследования - накопить информацию о среде и выяснить, какие действия приводят к полезным или выгодным результатам. Исследование особенно важно в начальной стадии обучения, когда агенту ещё неизвестны оптимальные стратегии или действия, и он должен исследовать среду, чтобы узнать больше о ней.

Эксплуатация (exploitation): Эксплуатация относится к стратегии, при которой агент использует свои текущие знания и опыт для принятия решений с целью получить максимальную награду или достичь определенных целей. Во время эксплуатации агент выбирает действия, которые, на основе его текущего знания о среде, считаются наиболее выгодными или оптимальными. Цель эксплуатации - использовать полученные знания для максимизации награды или достижения определенных целей. Эксплуатация может быть полезной, когда агент уже обладает значительным опытом и знаниями о среде.

Исследование и эксплуатация являются двумя взаимодополняющими аспектами RL. В начале обучения исследование имеет большую значимость, поскольку агенту нужно исследовать и изучить среду, чтобы найти оптимальные стратегии и действия. По мере накопления опыта и улучшения знаний, эксплуатация становится более важной, поскольку агент может использовать свои знания для принятия более обоснованных решений и достижения более высоких наград.

5.17. Объясните обучение с подкреплением на основе модели.

В процессе обучения, основанного на модели, агент предпринимает попытки оценивания вероятностей переходов T и вознаграждений R по выборкам, получаемым во время исследования, перед тем как использовать эти оценки для нахождения MDP решения.

Шаг 1: Эмпирическое обучение MDP модели:

- о Подсчёт числа исходов s' для каждой пары (s, a) ;
- о Нормализация числа исходов для получения оценки $T(s, a, s')$;
- о Оценка наград для каждого перехода (s, a, s') .

Шаг 2: Получение решения MDP: После схождения оценок T , обучение завершается генерацией политики (s) на основе алгоритмов итерации по значениям или политикам.

Обучение на основе модели интуитивно простое, но характеризуется большой пространственной сложностью (из-за необходимости хранения значений счетчиков переходов (s, a, s')).

5.18. Как классифицируются алгоритмы обучения с подкреплением без модели?

При обучении с подкреплением без модели используют три алгоритма, которые разделяются на две группы:

1. Алгоритмы пассивного обучения:
 - Алгоритм прямого оценивания;
 - Обучение на основе временных различий;
2. Алгоритмы активного обучения:
 - Q-обучение.

В случае пассивного обучения агент следует заданной политике и обучается ценностям состояний на основе накопления выборочных значений из эпизодов, что в общем, соответствует оцениванию политики при решении задачи MDP, когда T и R известны.

В случае активного обучения агент использует обратную связь для итеративного обновления его политики, пока не построит оптимальную политику после достаточного объема исследований.

5.19. Объясните алгоритм прямого оценивания (обучение без модели).

Цель: вычисление ценности каждого состояния при следовании политике π .

Идея: Усреднять наблюдаемые выборки значений ценности.

Алгоритм:

- о Действовать в соответствии с политикой π :

- о каждый раз при посещении состояния, подсчитывать и аккумулировать ценности состояния и число посещений состояния;

- о найти среднюю ценность состояния.

Положительные свойства прямого оценивания:

- о простота;

- о не требует никаких знаний T , R ;

- о вычисляет средние значения ценностей, используя просто выборки переходов.

Основным **недостатком** алгоритма являются значительные временные затраты.

5.20. Объясните алгоритм обучения на основе временных различий.

Основная идея TD-обучения (TD-temporal difference) заключается в том, чтобы обучаться на основе выборок при выполнении каждого действия [7, 8]:

- обновлять $V(s)$ каждый раз, при совершении перехода (s, a, s', r) ;
- более вероятные переходы будут вносить вклад в обновления более часто.

В ходе TD-обучения выполняется оценка ценности состояния при фиксированной политике. При этом ценность состояний вычисляется путем аппроксимации матожидания в уравнении Беллмана *экспоненциальным скользящим средним* выборочных значений $V(s)$:

- выборка $V(s)$:

$$sample = R(s, \pi(s), s') + \gamma V^\pi(s'), \quad (5.19)$$

- скользящее среднее выборок:

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + (\alpha)sample \quad (5.20)$$

или

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s)). \quad (5.21)$$

Правило обновления ценности состояния (5.21) называют **правилом обновления на основе временных различий**. Различие равно разности между выборочной наградой $(R + \gamma V(s'))$ и ожидаемой наградой $V(s)$, умноженной на скорость обучения α . Фактически эта разность есть погрешность, которую называют TD-погрешностью.

Последовательность действий **алгоритма TD-обучения** выглядит так:

1. Инициализировать $V(s)$ нулями или произвольными значениями;
2. Запустить эпизод, для каждого шага в эпизоде выполнить действие a в состоянии s , получить награду r и перейти в следующее состояние (s') ;
3. Обновить ценности состояний по правилу TD-обновления;
4. Повторять шаги 2 и 3, пока не будет достигнуто схождение ценности состояний.

Алгоритм обеспечивает более быстрое схождение ценности состояний по сравнению с алгоритмом прямого оценивания.

5.21. Объясните алгоритм q-обучения.

TD-обучение – подход к оцениванию политики без модели, моделирующий обновление Беллмана с помощью он-лайн усреднения выборок. Однако, если необходимо будет преобразовать значения $V^\pi(s)$ в новую политику возникнут сложности, так как для этого в соответствии с (5.15) необходимо оперировать значениями q-ценностей.

В q-обучении используется правило обновления, известное как правило итераций по q-ценностям [7]:

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right] \quad (5.22)$$

Это правило обновления следует из (5.10) и (5.9). Алгоритм q-обучения строится по схеме, аналогичной алгоритму TD-обучения, с использованием *выборок q-состояний*

$$sample = R(s, a, s') + \gamma \max_{a'} Q(s', a') \quad (5.23)$$

и их скользящем усреднении

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \cdot sample \quad (5.24)$$

или

$$Q(s, a) = Q(s, a) + \alpha(sample - Q(s, a)) = Q(s, a) + \alpha difference,$$

где разность $difference = sample - Q(s, a)$ рассматривается как отличие выборочной оценки q-ценности от ожидаемой оценки.

При достаточном объеме исследований и снижении в ходе обучения скорости обучения α оценки q-ценностей, получаемые с помощью (5.24), будут сходиться к оптимальным значениям для каждого q-состояния. В отличие от TD-обучения, которое требует применения дополнительных технологий извлечения политики, q-обучение формирует оптимальную политику непосредственно при выборе субоптимальных или случайных действий.

5.22. Объясните, что понимают под ϵ -жадной стратегией.

Жадная стратегия выбирает оптимальное действие среди уже исследованных. Что лучше искать: новое лучшее действие или действие, лучшее из всех исследованных действий? Это называется дилеммой между исследованием и эксплуатацией.

Чтобы разрешить дилемму, вводится ϵ -жадная стратегия: действие выбирается на основе текущей политики с вероятностью $1 - \epsilon$ и с вероятностью ϵ будет выполняться опробывание новых случайных действий (исследование). Значение ϵ должно уменьшаться со временем, поскольку заниматься исследованиями до бесконечности незачем. Таким образом, со временем политика переходит на эксплуатацию «хороших» действий.

5.23. Объясните алгоритм q-обучения с аппроксимацией.

5.24. Что понимают под функции разведки (исследования)?

ПРИЛОЖЕНИЕ А – ПОЛНЫЙ КОД РАБОЧИХ ФАЙЛОВ

Листинг 10 – Полный код файла analysis.py

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.0
    return answerDiscount, answerNoise

def question3a():
    answerDiscount = 0.4
    answerNoise = 0.0
    answerLivingReward = -2.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3b():
    answerDiscount = 0.5
    answerNoise = 0.3
    answerLivingReward = -1.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3c():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = -2.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3d():
    answerDiscount = 0.9
    answerNoise = 0.3
    answerLivingReward = -1.0

    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question3e():
    answerDiscount = 1.0
    answerNoise = 0.0
```



```

        answerLivingReward = 2.0
        return answerDiscount, answerNoise, answerLivingReward
        # If not possible, return 'NOT POSSIBLE'

def question8():
    answerEpsilon = None
    answerLearningRate = None
    return 'NOT POSSIBLE'

if __name__ == '__main__':
    print('Answers to analysis questions:')
    import analysis
    for q in [q for q in dir(analysis) if q.startswith('question')]:
        response = getattr(analysis, q)()
        print('  Question %s:\t%s' % (q, str(response)))

```

Листинг 11 – Код файла qlearningAgents.py

```

from game import *
from learningAgents import ReinforcementAgent
from featureExtractors import *

import random,util,math

class QLearningAgent(ReinforcementAgent):

    def __init__(self, **args):

        ReinforcementAgent.__init__(self, **args)
        self.QValue = util.Counter()

    def getQValue(self, state, action):

        return self.QValue[(state, action)]

    def computeValueFromQValues(self, state):

        best = float("-inf")

        actions = self.getLegalActions(state)
        for action in actions:

```

```

        if best < self.getQValue(state, action):
            best = self.getQValue(state, action)

    if best != float("-inf"):
        return best
    else:
        return 0.0

def computeActionFromQValues(self, state):

    best = float("-inf")
    act = []

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

    for action in actions:
        if best < self.getQValue(state, action):
            best = self.getQValue(state, action)
            act = [action]
        elif best == self.getQValue(state, action):
            act.append(action)

    return random.choice(act)

def getAction(self, state):

    legalActions = self.getLegalActions(state)

    if util.flipCoin(self.epsilon):
        return random.choice(legalActions)
    else:
        return self.computeActionFromQValues(state)

def update(self, state, action, nextState, reward):

    self.QValue[(state, action)] = (1-
self.alpha)*self.QValue[(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)

class PacmanQAgent(QLearningAgent):

    def __init__(self, epsilon=0.05,gamma=0.8,alpha=0.2, numTraining=0,
**args):

        args['epsilon'] = epsilon
        args['gamma'] = gamma
        args['alpha'] = alpha
        args['numTraining'] = numTraining
        self.index = 0
        QLearningAgent.__init__(self, **args)

    def getAction(self, state):

        action = QLearningAgent.getAction(self,state)
        self.doAction(state,action)
        return action

class ApproximateQAgent(PacmanQAgent):

    def __init__(self, extractor='IdentityExtractor', **args):
        self.feateExtractor = util.lookup(extractor, globals())()
        PacmanQAgent.__init__(self, **args)
        self.weights = util.Counter()

    def getWeights(self):
        return self.weights

    def getQValue(self, state, action):

        weights = self.getWeights()
        features = self.feateExtractor.getFeatures(state, action)
        q = weights * features
        return q

```

```

def update(self, state, action, nextState, reward):

    weights = self.getWeights()
    features = self.featsExtractor.getFeatures(state, action)

    nextActions = self.getLegalActions(nextState)
    maxNextQ = float("-inf")

    for act in nextActions:
        q = self.getQValue(nextState, act)
        maxNextQ = max(maxNextQ, q)

    if len(nextActions) == 0:
        maxNextQ = 0.0

    difference = reward + self.discount*maxNextQ -
self.getQValue(state, action)
    for feature in features:
        weights[feature] = weights[feature] + self.alpha * difference
    * features[feature]

def final(self, state):
    PacmanQAgent.final(self, state)

    if self.episodesSoFar == self.numTraining:

        pass

```

Листинг 12 – Код файла valueIterationAgents.py

```

import mdp, util

from learningAgents import ValueEstimationAgent
import collections

class ValueIterationAgent(ValueEstimationAgent):
    """
    * Пожалуйста, прочтите learningAgents.py перед тем, как читать
    это. *

    ValueIterationAgent принимает марковский процесс принятия
    решений

    (см. mdp.py) при инициализации и выполняет итерацию по значениям

```

для заданного количества итераций с использованием
коэффициента дисконтирования.

```
"""
def __init__(self, mdp, discount = 0.9, iterations = 100):
    """
    Ваш агент итераций по значениям должен принимать mdp при
    вызове конструктора, запустите его с указанным количеством
итераций,
    а затем действуйте в соответствии с полученной политикой.

    Некоторые полезные методы mdp, которые вы будете использовать:
        mdp.getStates() - возвращает список состояний MDP
        mdp.getPossibleActions(state) - возвращает кортеж возможных
действий в состоянии
        mdp.getTransitionStatesAndProbs(state, action)- возвращает
список из пар (nextState, prob) - s' и вероятности переходов T(s,a,s')
        mdp.getReward(state, action, nextState) - возвращает награду
R(s,a,s')
        mdp.isTerminal(state)- проверяет, является ли состояние
терминальным
    """
    self.mdp = mdp
    self.discount = discount
    self.iterations = iterations
    self.values = util.Counter() # Counter - словарь ценностей
состояний, по умолчанию содержит 0
    self.runValueIteration()

def runValueIteration(self):
    # Напишите код, реализующий итерации по значениям
    """ ВСТАВЬТЕ ВАШ КОД СЮДА """
    states = self.mdp.getStates()

    for i in range(self.iterations):
        temp = util.Counter()

        for state in states:
            best = float("-inf")
            actions = self.mdp.getPossibleActions(state)

            for action in actions:
```

```

        transitions = self.mdp.getTransitionStatesAndProbs(state,
action)

        sum = 0

        for transition in transitions:
            reward = self.mdp.getReward(state, action, transition[0])
            sum += transition[1]*(reward +
self.discount*self.values[transition[0]])

        best = max(best, sum)

        if best != float("-inf"):
            temp[state] = best

    for state in states:
        self.values[state] = temp[state]

def getValue(self, state):
    """
    Возвращает ценность состояния (вычисляется в __init__).
    """
    return self.values[state]

def computeQValueFromValues(self, state, action):
    """
    Вычисляет Q-ценность в состоянии по
    значению ценности состояния, сохраненному в self.values.
    """

    transitions = self.mdp.getTransitionStatesAndProbs(state,
action)

    sum = 0

    for transition in transitions:
        reward = self.mdp.getReward(state, action, transition[0])
        sum += transition[1]*(reward +
self.discount*self.values[transition[0]])

    return sum

def computeActionFromValues(self, state):

```

```

"""
    Определяет политику - лучшее действие в состоянии
    в соответствии с ценностями, хранящимися в self.values.

    Обратите внимание, что если нет никаких допустимых действий,
    как в случае терминального состояния, необходимо вернуть None.
"""
*** ВСТАВЬТЕ ВАШ КОД СЮДА ***

best = float("-inf")
act = None
actions = self.mdp.getPossibleActions(state)
for action in actions:
    q = self.computeQValueFromValues(state, action)
    if q > best:
        best = q
        act = action

return act

def getPolicy(self, state):
    return self.computeActionFromValues(state)

def getAction(self, state):
    "Возвращает политику в состоянии (без исследования)"
    return self.computeActionFromValues(state)

def getQValue(self, state, action):
    return self.computeQValueFromValues(state, action)

class AsynchronousValueIterationAgent(ValueIterationAgent):
    """
        * Пожалуйста, прочтите learningAgents.py перед тем, как читать
        это. *

        AsynchronousValueIterationAgent принимает марковский процесс
        принятия решений
        (см. mdp.py) при инициализации и выполняет итерацию по значениям
        для заданного количества итераций с использованием
        коэффициента дисконтирования.
    """
    def __init__(self, mdp, discount = 0.9, iterations = 1000):
        """

```

Ваш агент итераций по значениям должен принимать mdp при вызове конструктора, запустите его с указанным количеством итераций,

а затем действуйте в соответствии с полученной политикой. Каждая итерация обновляет значение только одного состояния, которое циклически выбирается из списка состояний. Если выбранное состояние является конечным, на этой итерации ничего не происходит.

Некоторые полезные методы mdp, которые вы будете использовать:

- mdp.getStates() - возвращает список состояний MDP
- mdp.getPossibleActions(state) - возвращает кортеж возможных действий в состоянии
- mdp.getTransitionStatesAndProbs(state, action)- возвращает список из пар (nextState, prob) - s' и вероятности переходов $T(s, a, s')$
- mdp.getReward(state, action, nextState) - возвращает награду $R(s, a, s')$
- mdp.isTerminal(state)- проверяет, является ли состояние терминальным

```
"""
ValueIterationAgent.__init__(self, mdp, discount, iterations)

def runValueIteration(self):
    """ ВСТАВЬТЕ ВАШ КОД СЮДА """
    states = self.mdp.getStates()
    for i in range(self.iterations):
        v = self.values.copy()
        s0 = states[i % len(states)]
        if self.mdp.isTerminal(s0):
            continue
        v[s0] = -float("inf")
        for a in self.mdp.getPossibleActions(s0):
            val = 0
            for s, p in self.mdp.getTransitionStatesAndProbs(s0, a):
                val += p * (self.mdp.getReward(s0, a, s) +
self.discount * self.values[s])
            v[s0] = max(v[s0], val)
        self.values = v.copy()
```



```

class
PrioritizedSweepingValueIterationAgent(AsynchronousValueIterationAgent):
    """
    * Пожалуйста, прочтите learningAgents.py перед тем, как читать
    это. *

    Агент PrioritizedSweepingValueIterationAgent принимает
    марковский процесс принятия решения (см. Mdp.py) при инициализации и
    выполняет итерации по значениям с разверткой приоритетных состояний при
    заданном числе итераций с использованием предоставленных
    параметров.
    """
    def __init__(self, mdp, discount = 0.9, iterations = 100, theta = 1e-
5):
        """
        Ваш агент итерации по развертке приоритетных значений должен
        принимать на вход МДП при создании, выполнять заданное количество
        итераций,
        а затем действовать в соответствии с полученной политикой.
        """
        self.theta = theta
        ValueIterationAgent.__init__(self, mdp, discount, iterations)

    def runValueIteration(self):

        """ ВСТАВЬТЕ ВАШ КОД СЮДА """
        pred = self.generatePredecessors()
        pQ = util.PriorityQueue()
        states = self.mdp.getStates()
        for s in states:
            if self.mdp.isTerminal(s):
                continue
            pQ.push(s, -self.getDiff(s))
        for i in range(self.iterations):
            if pQ.isEmpty():
                break
            v = self.values.copy()
            s0 = pQ.pop()
            v[s0] = -float("inf")
            for a in self.mdp.getPossibleActions(s0):
                val = 0

```

