

ЛАБОРАТОРНАЯ РАБОТА № 5

«ИССЛЕДОВАНИЕ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ»

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.

Постановка задачи

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 значениями политикам, а также алгоритмы обучения подкреплением с моделями и без моделей: алгоритм прямого обучения, алгоритм TD-обучения, алгоритм Q-обучения.

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.

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
```

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 `Rasman`, 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.

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 со стояний `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).

5. 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.

6. 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.

7. 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`.

8. В задании 5 необходимо реализовать алгоритм итераций по приоритетным значениям.

При реализации алгоритма следуйте указаниям, сформулированным в самом задании.

8. 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).

9. 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)`.

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

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

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

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 автооцениватель:

```
python3.11 autograder.py -q q1
```

Подсказка: в среде BookGrid, используемой по умолчанию, при выполнении 5 итераций по значениям должен получиться результат, изображенный на рисунке 5.3:

```
python3.11 gridworld.py -a value -i 5
```

Разработанный код представлен ниже:

```
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
# Counter - словарь ценностей состояний, по умолчанию содержит
0
self.values = util.Counter()
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)

```

```

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

Finished at 21:50:01

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

```

Рисунок 1 – Результаты автооценителя



Рисунок 2 – Результат в среде BookGrid

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 (рисунок 5.4).

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:

```
python3.11 gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

Разработанный код представлен ниже:

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

```
### Question q2: 1/1 ###  
  
Finished at 22:07:08  
  
Provisional grades  
=====
```

Question q2: 1/1

Total: 1/1

Рисунок 3 – Результаты автооценителя



Рисунок 4 – Результат вызова

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. Эти пути обозначены зеленой стрелкой на рисунке 5.5.

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 автооцениватель:

```
python3.11 autograder.py -q q3
```

Разработанный код представлен ниже:

```
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'
```

```
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 22:25:17

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

Рисунок 5 – Результаты автооценивателя

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.

```
python3.11 autograder.py -q q4
```

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

```
python3.11 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).

Разработанный код представлен ниже:

```
def runValueIteration(self):
    # Напишите код, реализующий асинхронные итерации по значениям
    """ 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()
```

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

Finished at 22:27:11

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

Рисунок 6 – Результаты автооценителя



Рисунок 7 – Результаты вычисления политики



Рисунок 8 – Результаты вычисления Q-значений

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 $-\text{diff}$ (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 $0, 1, 2, \dots, \text{self.iterations} - 1$ 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 $\text{diff} > \theta$, 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 *python3.11 autograder.py -q q5*

Вы можете запустить PrioritizedSweepingValueIterationAgent в Gridworld, используя следующую команду.

```
python3.11 gridworld.py -a priosweepvalue -i 1000
```

Разработанный код представлен ниже:

```
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

```

```

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 22:32:20

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

```

Рисунок 9 – Результаты автооценителя



Рисунок 10 – Выполнение агента в среде Gridworld

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:

```
python3.11 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 Расмана 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 автооценщик:

python3.11 autograder.py -q q6

Разработанный код представлен ниже:

```
class QLearningAgent(ReinforcementAgent):
    """
    Агент с Q-обучением

    Функции, которые необходимо дополнить:
    - computeValueFromQValues
    - computeActionFromQValues
    - getQValue
    - getAction
    - update

    Переменные, к которым у вас есть доступ
    - self.epsilon (вероятность исследования)
    - self.alpha (скорость обучения)
    - self.discount (коэффициент дисконтирования)

    Функции, которые Вам следует использовать
    - self.getLegalActions(state)
      возвращает допустимые действия в состоянии state
    """

    def __init__(self, **args):
        """Здесь можно инициализировать Q-ценности"""
        ReinforcementAgent.__init__(self, **args)

        """ ВСТАВЬТЕ ВАШ КОД СЮДА """
        self.QValue = util.Counter()

    def getQValue(self, state, action):
        """
        Возвращает Q(state,action)
        Должен вернуть 0.0, если состояние никогда не встречалось
        или Q-ценность в ином случае
        """
        """ ВСТАВЬТЕ ВАШ КОД СЮДА """

        return self.QValue[(state, action)]

    def computeValueFromQValues(self, state):
        """
        Возвращает ценность состояния state
        путем вычисления max_action Q(state,action)
        где максимум ищется по всем допустимым действиям.
        Если нет допустимых действий,
        что имеет место в терминальных состояниях,

```


метод должен вернуть значение 0.0

```
"""
*** ВСТАВЬТЕ ВАШ КОД СЮДА ***

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):
    """
```

Вычисляет лучшее действие, которое нужно предпринять в состоянии.
Обратите внимание, если нет допустимых действий, что имеет место
в терминальных состояниях, метод должен вернуть None.
"""

```
*** ВСТАВЬТЕ ВАШ КОД СЮДА ***

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):
    """
```

Возвращает действие, которое нужно предпринять в текущем
состоянии.

С вероятностью self.epsilon предпринимается случайное действие
и в противном случае предпринимается наилучшее действие.
Обратите внимание, что если нет никаких допустимых действий,
что имеет место в терминальном состоянии, вы должны
вернуть в качестве действия None.

ПОДСКАЗКА: вы можете использовать util.flipCoin(prob)

ПОДСКАЗКА: для случайного выбора из списка используйте
random.choice (list)

```

"""

*** ВСТАВЬТЕ ВАШ КОД СЮДА ***

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):
    """

    Родительский класс вызывает этот метод, чтобы выполнить
    q-обновление в соответствии с переходом
    state => action => nextState = .reward.
    Вы должны реализовать здесь q-обновление.

    ПРИМЕЧАНИЕ: вы никогда не должны вызывать этот метод

    """

    *** ВСТАВЬТЕ ВАШ КОД СЮДА ***

    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)

```

```

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

Finished at 22:41:05

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

```

Рисунок 11 – Результаты автооценителя

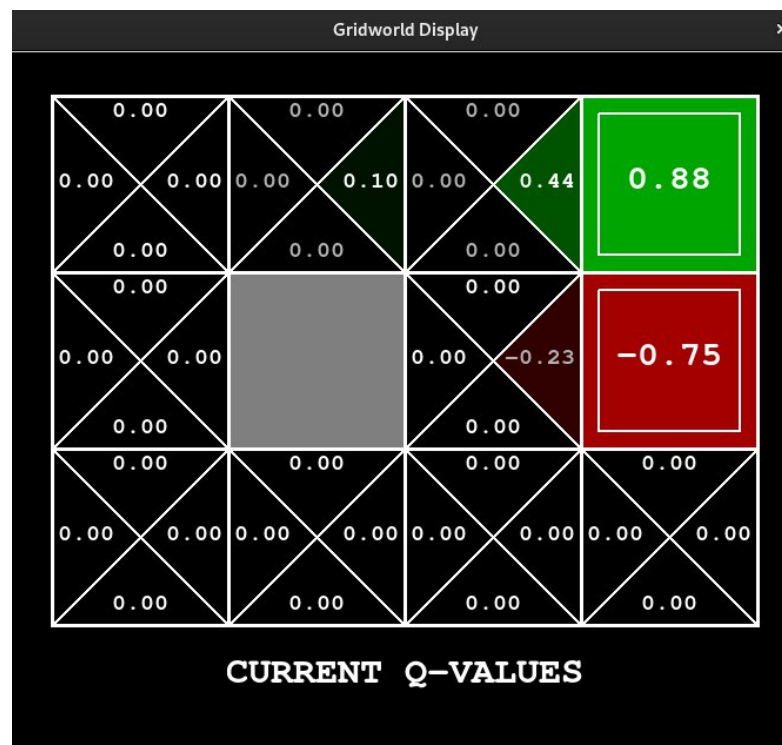


Рисунок 12 – Результаты Q-обучения

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`).

```
python3.11 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 значений ϵ (параметр `-e`). Соответствует ли такое поведение агента вашим ожиданиям?

```
python3.11 gridworld.py -a q -k 100 --noise 0.0 -e 0.1
```

```
python3.11 gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

Чтобы протестировать вашу реализацию, запустите автооцениватель:

```
python3.11 autograder.py -q q7
```

Теперь без дополнительного кодирования вы сможете запустить q -обучение для робота `crawler` (рисунок 5.7):

```
python3.11 crawler.py
```

Если вызов не сработает, 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 эpsilon являются параметрами алгоритма обучения, а коэффициент дисконтирования является свойством среды.

Разработанный код представлен ниже:

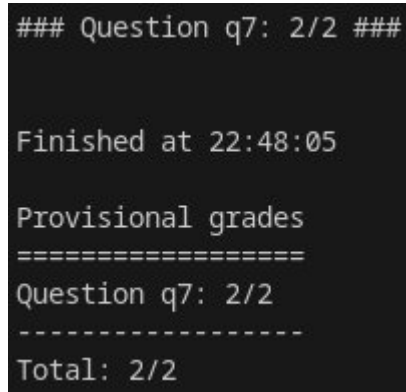
```
def getAction(self, state):
    """
    Возвращает действие, которое нужно предпринять в текущем
    состоянии.
    С вероятностью self.epsilon предпринимается случайное действие
    и в противном случае предпринимается наилучшее действие.
    Обратите внимание, что если нет никаких допустимых действий,
    что имеет место в терминальном состоянии, вы должны
    вернуть в качестве действия None.

    ПОДСКАЗКА: вы можете использовать util.flipCoin(prob)
    ПОДСКАЗКА: для случайного выбора из списка используйте
    random.choice (list)
    """

    """* ВСТАВЬТЕ ВАШ КОД СЮДА """

    legalActions = self.getLegalActions(state)

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



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

Finished at 22:48:05

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

Рисунок 13 – Результаты автооценителя



Рисунок 14 – Окончательные значения q-ценностей

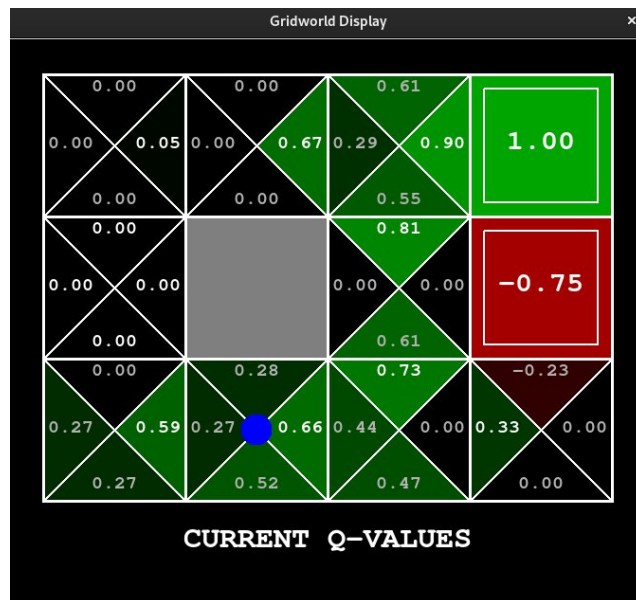


Рисунок 15 – Поведение агента при эпсилон 0.1

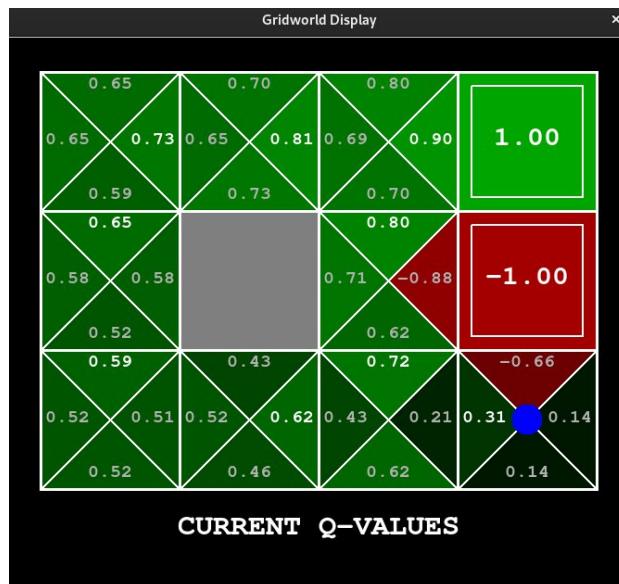


Рисунок 16 – Поведение агента при эпсилон 0.9

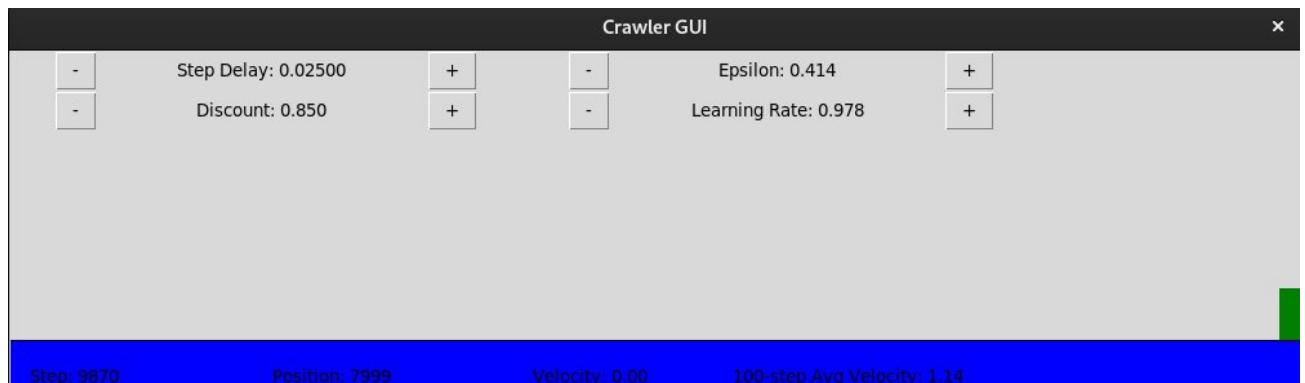


Рисунок 17 – Q-обучение для робота crawler

Задание 8 (1 балл). Переход по мосту98

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:

```
python3.11 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 выборе 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 автооцениватель:

```
python3.11 autograder.py -q q8
```

Разработанный код представлен ниже:

```
def question8():
    answerEpsilon = None
    answerLearningRate = None
    return 'NOT POSSIBLE'
    # If not possible, return 'NOT POSSIBLE'
```

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

Finished at 23:05:23

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

Рисунок 18 – Результаты автооценивателя



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:

```
python3.11 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 наблюдаемо действие может быть оптимальным. Остерегайтесь применения функции `argmax` класса `util.Counter`.

Чтобы оценить задание 9, выполните:

```
python3.11 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:

```
python3.11 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 Пакман 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 Пакман 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).

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 `PacmanQAgent`.

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:

```
python3.11 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:

```
python3.11 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):

```
python3.11 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:

```
python3.11 autograder.py -q q10
```

Разработанный код представлен ниже:

```
class ApproximateQAgent(PacmanQAgent):  
    """  
        Агент Q-обучения с аппроксимацией  
  
        Вам нужно только переопределить getQValue  
        и update. Все другие функции QLearningAgent  
        должны работать без изменения.  
        Также, при желании, допишите метод final(self, state),  
        чтобы отображать итоговые значения весов признаков и  
        такие параметры как: alpha, epsilon, gamma,  
        число эпизодов обучения.  
    """  
  
    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):  
        """  
            Должен возвращать Q(state,action) = w * featureVector  
            где * оператор матричного умножения  
        """  
  
        """ ВСТАВЬТЕ ВАШ КОД СЮДА """
```

```

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

def update(self, state, action, nextState, reward):
    """
    Обновляет веса признаков на основе данных переходов (s,a,s',r)
    """

    """ ВСТАВЬТЕ ВАШ КОД СЮДА """

    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):
    "Вызывается в конце игры"

    # вызов метода final супер-класса
    PacmanQAgent.final(self, state)

    # проверяем, завершилось ли обучение?
    if self.episodesSoFar == self.numTraining:
        # здесь, если захотите, вы можете распечатать веса при отладке

        """ ВСТАВЬТЕ ВАШ КОД СЮДА """

    pass

```

```

Training Done (turning off epsilon and alpha)
-----
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 495
Pacman died! Score: -514
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Average Score: 397.7
Scores:      495.0, 499.0, 495.0, 495.0, 503.0, 495.0, -514.0, 503.0, 503.0, 503.0
Win Rate:    9/10 (0.90)
Record:      Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win

```

Рисунок 22 – Результат работы агента ApproximateQAgent

```

Beginning 50 episodes of Training
Training Done (turning off epsilon and alpha)
-----
Pacman emerges victorious! Score: 525
Pacman emerges victorious! Score: 525
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Average Score: 528.2
Scores:      525.0, 525.0, 529.0, 529.0, 529.0, 529.0, 529.0, 529.0, 529.0, 529.0
Win Rate:    10/10 (1.00)
Record:      Win, Win, Win, Win, Win, Win, Win, Win, Win, Win

```

Рисунок 23 – Результат работы агента основанного на аппроксимации состояний с использованием SimpleExtractor

```

Beginning 50 episodes of Training
Training Done (turning off epsilon and alpha)
-----
Pacman emerges victorious! Score: 1344
Pacman emerges victorious! Score: 1318
Pacman emerges victorious! Score: 1322
Pacman emerges victorious! Score: 1543
Pacman emerges victorious! Score: 1333
Pacman emerges victorious! Score: 1322
Pacman emerges victorious! Score: 1339
Pacman emerges victorious! Score: 1331
Pacman emerges victorious! Score: 1346
Pacman emerges victorious! Score: 1335
Average Score: 1353.3
Scores:      1344.0, 1318.0, 1322.0, 1543.0, 1333.0, 1322.0, 1339.0, 1331.0, 1346.0, 1335.0
Win Rate:    10/10 (1.00)
Record:      Win, Win, Win, Win, Win, Win, Win, Win, Win, Win

```

Рисунок 24 – Результат работы агента на более сложном поле

```
### Question q10: 3/3 ###  
  
Finished at 23:27:26  
  
Provisional grades  
=====  
Question q10: 3/3  
-----  
Total: 3/3
```

Рисунок 25 – Результаты работы автооценителя

ВЫВОДЫ

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

ПРИЛОЖЕНИЕ А

Код программы