Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
«Севастопольский государственный университет»

Программирование на языке Python для систем искусственного интеллекта

Методические указания
к лабораторным работам по дисциплине
«Методы и системы искусственного интеллекта»
для студентов дневной и заочной форм обучения направлений:
09.03.02 — «Информационные системы и технологии»,
09.03.03 — «Прикладная информатика»

Севастополь
2025

УДК 004.89:004.43(076.5)
ББК 32.813:32.973-018.1я73
П784

Рецензенты:
Чернега В. С., канд. техн. наук, доцент кафедры Информационных систем.
Брюховецкий А. А., канд. техн. наук, доцент кафедры Информационные технологии и компьютерных системы

Составитель: В. Н. Бондарев

П784 Программирование на языке Python для систем искусственного интеллекта : методические указания к лабораторным работам по дисциплине «Методы и системы искусственного интеллекта» для студентов дневной и заочной форм обучения направлений: 09.03.02 — «Информационные системы и технологии», 09.03.03 – «Прикладная информатика» / Севастопольский государственный университет ; сост. В. Н. Бондарев. — Севастополь : СевГУ, 2025. —150 с.

Методические указания предназначены для использования в процессе выполнения лабораторных работ по дисциплине “Методы и системы искусственного интеллекта” для студентов дневной и заочной форм обучения направлений: 09.03.02 — “Информационные системы и технологии”, 09.03.03 – “Прикладная информатика”. Излагаются теоретические сведения, необходимые для выполнения лабораторных работ, содержатся задания для выполнения, программа исследований и порядок выполнения заданий, требования к содержанию отчета, контрольные вопросы.

УДК 004.89:004.43(076.5)
ББК 32.813:32.973-018.1я73

Методические указания рассмотрены и утверждены на заседании кафедры «Информационные системы» (протокол № *8* от *16 апреля* 20*24* г.)

Методические указания рассмотрены и рекомендованы к изданию на заседании Учёного Совета Института информационных технологий, протокол
№ *05* от *18 апреля* 20*24* г.

Ответственный за выпуск: заведующий кафедрой Информационных систем, канд. физ.-мат. наук, доцент И. П. Шумейко

Издательский номер № /2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Лабораторные работы по дисциплине «Методы и системы искусственного интеллекта», представленные в настоящих методических указаниях, направлены на освоение основных идей и методов, лежащих в основе интеллектуальных компьютерных систем. Основное внимание уделяется алгоритмам поиска решений задач в пространстве состояний, алгоритмам эвристического поиска, алгоритмам мультиагентного поиска, вероятностному выводу в условиях неопределенности, алгоритмам машинного обучения с учителем и с подкреплением.

При выполнении лабораторных работ используется игровая среда AI Pacman, разработанная Джоном ДеНеро и Дэном Кляйном специально для обучения искусственному интеллекту [6]. Оригинальная видео игра Pac-Man была разработана японской компанией Namco в1980 году. Все лабораторные работы реализуются на языке Python – одном из самых распространённых языков программирования, широко используемом в современном искусственном интеллекте.

К концу лабораторного курса вы приобретёте навыки разработки на языке Python автономных агентов, которые эффективно принимают решения в полностью информированных, частично наблюдаемых и враждебных условиях. Ваши агенты будут делать выводы в неопределенных условиях и оптимизировать свои действия для получения максимума ожидаемой награды. Методы, которые вы изучите в этом курсе, применимы к широкому спектру проблем искусственного интеллекта и послужат основой для дальнейшего изучения иных прикладных областей современного искусственного интеллекта.

1. ЛАБОРАТОРНАЯ РАБОТА № 1
«ИССЛЕДОВАНИЕ БАЗОВЫХ ФУНКЦИЙ ЯЗЫКА PYTHON»

1.1. Цель работы

Изучение технологии подготовки и выполнения программ на языке Python, исследование свойств функций языка Python, используемых при обработке последовательностей, формирование навыков определения классов языка Python

1.2. Краткие теоретические сведения

1.2.1. Python

Python — динамически типизированный, мультипарадигменный язык программирования, официально опубликованный в 1991 году. Разработан Гвидо ван Россумом (Guido van Rossum) в Национальном исследовательском институте математики и компьютерных наук (г. Амстердам).

Python быстро стал одним из самых популярных языков программирования. В настоящее время является одним из основных языков обработки данных и искусственного интеллекта.

1.2.2 Установка дистрибутива Anaconda и загрузка задания

В лабораторных работах рекомендуется использовать дистрибутив Anaconda Python, отличающийся простотой установки. В него входит практически всё необходимое для выполнения лабораторных работ, в том числе:

- интерпретатор IPython;
- большинство библиотек Python;
- локальный сервер Jupyter Notebook для загрузки и выполнения блокнотов;
- интегрированная среда разработки Spyder IDE (IDE - Integrated Development Environment).

Все задания из лабораторных работ протестированы в версии Python 3.9.

Программу установки Python с помощью Anaconda для Windows, macOS и Linux можно загрузить по адресу: https://www.anaconda.com/download/

Когда загрузка завершится, запустите программу установки. Чтобы установленная копия Anaconda работала правильно, не перемещайте ее файлы после установки.

Если в вашей системе уже была установлена иная версия Python, то рекомендуется создать отдельную среду для работы с Python 3.9 (можно использовать версии Python 3.9-3.11), введя в командном окне conda команду:

```
conda create --name <env-name> python=3.9
```

Здесь <env-name> - название среды, например, python39. Чтобы войти в среду, которая была только что создана, активируйте её:

```
conda activate python39
```

Проверьте используемую в среде версию Python, набрав в команду:

```
python –V
   Python 3.9.7
```

Установите в созданную среду Spyder IDE командой: conda install spyder. Также установите библиотеки для работы с динамическими массивами numpy и визуализации данных matplotlib:

```
conda install numpy
conda install matplotlib
```

Среда готова для работы. Для выхода из среды python39 выполните команду деактивации

```
conda deactivate
```

Вы вернетесь к версии Python, которая была ранее установлена в системе.

Все файлы, которые содержат задания данной лабораторной работы, находятся архиве: **МиСИИ_лаб1_2024.zip**. Файл можно получить у преподавателя или найти по адресу размещения электронных ресурсов дисциплины в системе Moodle. Создайте локальную рабочую папку, откройте её и загрузите все файлы из указанного архива

1.2.3. Вызов интерпретатора Python

Python можно запускать в одном из двух режимов. Его можно использовать интерактивно, работая с интерпретатором, или вызвать из командной строки для выполнения сценария.

Покажем, как использовать интерпретатор Python в интерактивном режиме. Откройте окно командной строки в своей системе и запустите **командную строку Anaconda** из меню Пуск. В окне командной строки введите команду **python** и нажмите Enter. На экране появится сообщение, подобное изображенноиу на рисунке 1.1. (зависит от платформы и версии Python):

```
(base) C:\Users\User>python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc
. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Рисунок 1.1 – Экранное сообщение при выполнении команды python

Строка с символами ">>>" на рисунке1.1 обозначает *приглашение*, означающее, что Python ожидает ввода. Чтобы выйти из интерактивного режима введите команду exit().

Вы также можете работать в интерактивном режиме, используя непосредственно оболочку IPython. Для этого вместо вышеупомянутой команды python следует ввести команду ipython. На экране появится сообщение, подобное изображенному на рисунке 1.1, но строка приглашения теперь будет в виде “In[1]”.

Среди функций, которые предоставляет IPython, наиболее интересны следующие:

динамический анализ объектов;

доступ к оболочке системы через командную строку;

прямая поддержка профилирования;

работа с отладочными средствами.

IPython является частью более крупного проекта под названием Jupyter, предоставляющего возможность работы с интерактивными блокнотами через интернет-браузер.

Также возможна работа с IPython в интегрированной среде Spyder IDE, которая предоставляет традиционный спектр возможностей, свойственных IDE. При этом консольное окно IPython обычно располагается внизу справа основного экрана Spyder IDE (рисунок 1.2).

Ниже будут приведены примеры команд для консоли Python. Команды консоли IPython аналогичны, но требуют ввода команд, исполняемых операционной средой, начиная со знака «!», например:

```
!python -V
```

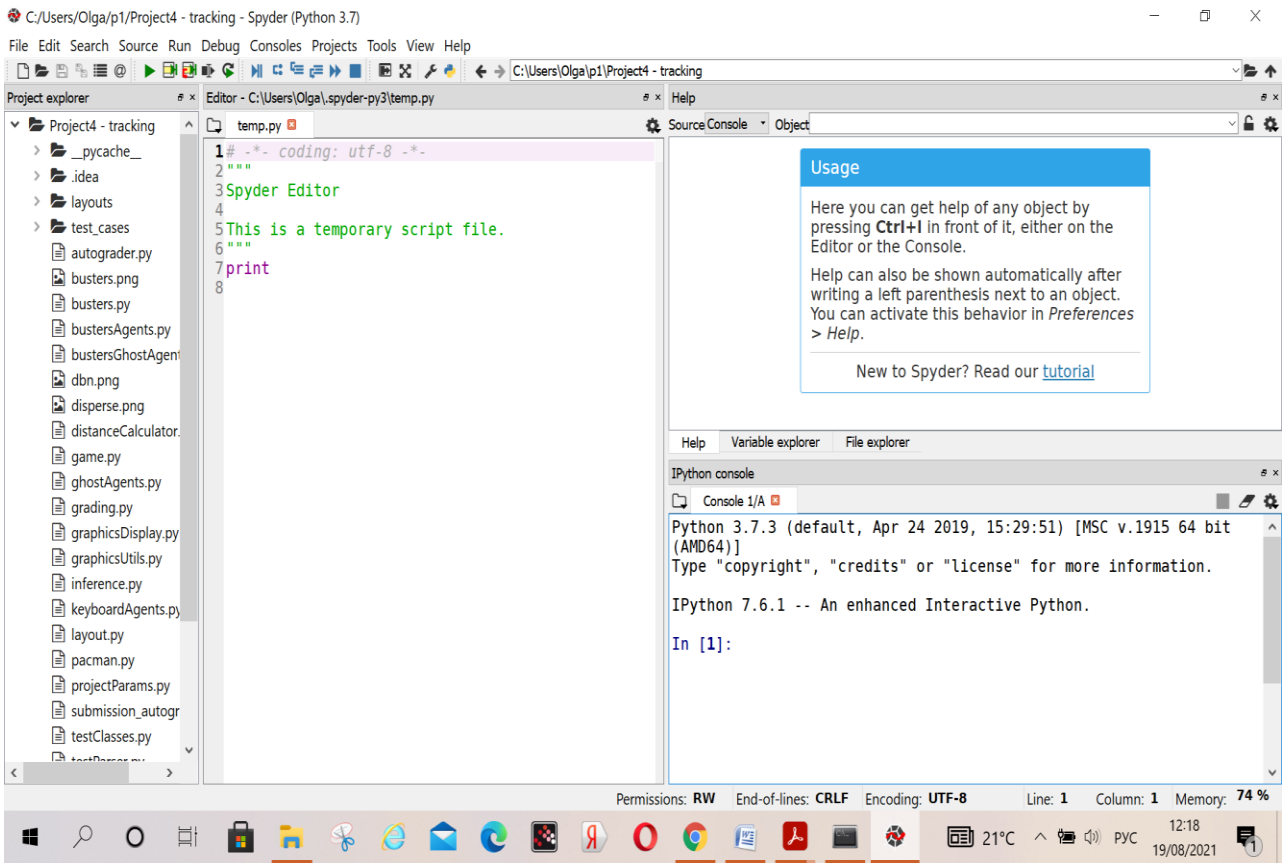


Рисунок 1.2 – Основной экран интегрированной среды Spyder IDE

1.2.4. Введение в программирование на языке Python

Арифметические и логические операторы

Интерпретатор Python можно использовать для вычисления значений арифметических выражений. Если вы введёте арифметическое выражение после приглашения ">>>", то оно будет вычислено, а результат будет возвращен в следующей строке:

```
>>>2+3
5
>>>2*3
6
>>> num = 8.0                # присваивание
>>> num + = 2.5              # инкремент
>>> print (num)              # печать
10.5
```

Python поддерживает работу с различными типами числовых данных: int, float, complex. Чтобы получить информацию о типе значения переменной, воспользуйтесь встроенной *функцией* Python type:

```
>>> type(num)
float
```

В таблице 1.1 перечислены арифметические операторы Python.

Таблица 1.1 - Арифметические операторы Python

Операция Python	Арифметический оператор	Алгебраическое выражение	Выражение Python
Сложение	+	$f + 7$	`f + 7`
Вычитание	–	$p - c$	`p - c`
Умножение	*	$b \cdot m$	`b * m`
Возведение в степень	**	x^y	`x ** y`
Деление	/	x/y, или $\frac{x}{y}$, или $x \div y$	`x / y`
Целочисленное деление	//	$[x/y]$, или $\left\lfloor \frac{x}{y} \right\rfloor$, или $[x \div y]$	`x // y`
Остаток от деления	%	$r \bmod s$	`r % s`

В Python также существуют логические операторы и операции отношений, выполнящие или возвращающие результаты действий с логическими значениями типа bool (True и False) :

```
>>>1==0                    # равно
False
>>>1!=0                    # не равно
True
>>> not (1==0)             # логическое отрицание
True
>>> (2==2) and (2==3)      # логическое И
False
>>> (2==2) or (2==3)       # логическое ИЛИ
True
```

Строки

Python имеет встроенный строковый тип (класс) str. Сроковая константа заключается в одинарные или двойные кавычки: ‘hello’, “hello”. Оператор “+” позволяет выполнить конкатенацию строк:

```
>>>’искусственный’+”интеллект”
’искусственныйинтеллект’
```

Существует много полезных встроенных методов для работы со строками, например:

```
>>> 'artificial'.upper ()
'ARTIFICIAL'
>>> 'HELP'.lower ()
'help'
>>> len('Help')
4
```

Строки являются *неизменяемыми* — это означает, что их нельзя модифицировать в программе. Вы можете прочитать отдельные символы в строке, но при попытке присвоить новое значение одному из символов строки будет ошибка TypeError:

```
>>> s = 'hello'
>>> s[0]
'h'
s[0] = 'H'
---------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-15-812ef2514689> in <module>()
----> 1 s[0] = 'H'
TypeError: 'str' object does not support item assignment
```

Python позволяет сохранять строки (строковые выражения) в переменных:

```
>>> hw='%s %s %d' % ('hello', 'world', 11)          #форматирование вывода
```

```
>>> print(hw)
hello world 11
>>> hw.replace('l','ell')                    #замена символов
'heellello worelld 11'
```

Если в строке встречается обратный слеш (\), то он является *управляющим символом*, а в сочетании с непосредственно следующим за ним символом образует *управляющую последовательность* (\n, \t , \\, \',\"):

```
>>> print('Welcome\nto\n\nPython!')       # \n – управляющий символ новой строки
Welcome
to

Python!
```

В Python имеются строки в тройных кавычках. Используйте их для создания:
многострочных строк;
строк, содержащих одинарные или двойные кавычки;
doc-строк — рекомендуемого способа документирования некоторых компонентов программы. Например:

```
>>> print("""Display "hi" and 'bye' in quotes""")
Display "hi" and 'bye' in quotes
```

Python предоставляет возможности форматирования значений с использованием форматных строк (*f - строк*).

```
>>> average = 15.6666
>>> print(f'Среднее равно {average:.2f}')
Среднее равно 15.67
```

Буква f перед открывающей кавычкой строки означает, что это форматная строка. Чтобы указать, где должно вставляться выводимое значение, используйте поля в фигурных скобках { }. Поле с переменной {average} преобразует значение переменной average в строковое представление, а затем меняет {average} *заменяющим текстом*. В форматной строке за переменной (выражением) в заполнителе может следовать двоеточие (:) и *форматный спецификатор*, который описывает, как должен форматироваться заменяющий текст. Форматный спецификатор .2f форматирует average при выводе как число с плавающей точкой (f) с двумя цифрами в дробной части (.2). Выражения в фигурных скобках (в поле) могут содержать значения, переменные или другие выражения (например, вычисления или вызовы функций).

В приведенном примере спецификатор ‘.2f’ определял тип представления переменной. Существуют и другие типы представлений (b, o, x и X), форматирующие целые числа для представления в двоичной, восьмеричной и шестнадцатеричной системах счисления.

В f-строках можно задавать ширину поля вывода и выполнять выравнивание. Например:

```
>>> f'[{3.5:<15f}]'                    # выравнивание слева в поле из 15 позиций
'[3.500000       ]'
>>> f'[{"hello":>15}]'                 # выравнивание справа в поле из 15 позиций
'[          hello]'
```

Форматные строки Python были добавлены в язык в версии 3.6. До этого форматирование строк выполнялось с помощью метода format. Собственно, функциональность форматных строк основана на средствах метода format. Метод format может вызываться для *форматной строки*, содержащей *заполнители* в фигурных скобках { }:

```
'{:.2f}'.format(14.989)
'14.99'
```

. Чтобы узнать, какие методы Python предоставляет для работы с тем или иным типом данных, используйте команды dir и help. Например, методы для работы со строками:

```
>>>s='abc'
>>>dir(s)
```

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

```
>>>help(s.find)

Help on built-in function find:

find(...) method of builtins.str instance
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

Функция find возвращает наименьший индекс s, начиная с которого в s найдена подстрока sub. Необязательные аргументы start и end интерпретируются как обозначения сегмента строки для поиска:

```
>>>s.find('b')
1
```

Контейнеры: списки, кортежи, словари, множества

Python оснащен некоторыми полезными встроенными структурами данных – списками, кортежами, словарями, множествами.

Списки

В списках (класс list) хранится последовательность изменяемых элементов:

```
>>> fruits = ['apple', 'orange', 'pear', 'banana']   # задание списка
>>> fruits[0]                                        # обращение по индексу
'apple'
```

Можно создать список из целых чисел с помощью функций range() и list():

```
>>> nums=list(range(5))    #range формирует последовательность от 0 до 4
>>> print(nums)
[0, 1, 2, 3, 4]
```

Здесь вызов функции range(5) создает итерируемый объект, который представляет последовательность целых чисел от 0 и до значения аргумента 5, *не включая* последний, в данном случае 0, 1, 2, 3, 4. Функция list создает из последовательности чисел список [0, 1, 2, 3, 4]

Для *объединения* списков можно использовать оператор сложения (+):

```
>>> otherFruits = ['kiwi', 'strawberry']
>>> fruits + otherFruits
>>> ['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']
```

Обращение к элементам списков выполняется с помощью индексов в квадратных скобках: fruits[0] – первый элемент списка. Python также допускает отрицательную индексацию с конца списка. Например, fruits[-1] обращается к последнему элементу 'banana', а

```
>>> fruits[-2]                          #2-ой с конца
'pear'
```

вернет второй элемент списка с его конца.

Можно *индексировать несколько смежных элементов* одновременно с помощью операторов среза (сегментирования). Например, fruits[1:3] возвращает список, содержащий элементы в позициях 1 и 2. В общем, fruits[start: stop] вернет элементы в позициях start, start + 1, ..., stop-1. Обращение fruits[start:] вернет все элементы списка, начиная с индекса start. Также fruits[:end] вернет все элементы с начала списка и до элемента в позиции end. Примеры:

```
>>> fruits=['apple', 'orange', 'pear', ' banana']
>>> fruits[0:2]
['apple', 'orange']
>>> fruits[:3]
['apple', 'orange', 'pear']
>>> fruits[2:]
['pear', 'banana']
>>> len(fruits)
4
```

Для работы со списками в Python имеется большой набор встроенных методов. Например,

```
>>> fruits.pop()                          #удаляет последний эл-т и возвращает его
'banana'
>>> fruits
['apple', 'orange', 'pear']                #состояние списка после pop
>>> fruits.append('grapefruit')            # добавляет в конец списка
>>> fruits
['apple', 'orange', 'pear', 'grapefruit']  #состояние списка после добавления
>>> fruits[-1] = 'pineapple'               #заменить последний эл-т
>>> fruits
['apple', 'orange', 'pear', 'pineapple']   #состояние после замены
>>> fruits. insert(0, ‘grapefruit’)        #вставка эл-та в заданную позицию
>>> fruits
['grapefruit', 'apple', 'orange', 'pear', 'pineapple']
```

Элементы, хранящиеся в списках, могут быть любого типа. Так, например, элементами списка могут быть списки:

```
>>> lstOfLsts = [['a', 'b', 'c'], [1, 2, 3], ['one', 'two', 'three']]
>>> lstOfLsts[1][2]
3
>>> lstOfLsts[0].pop()
'c'
>>> lstOfLsts
[['a', 'b'], [1, 2, 3], ['one', 'two', 'three']]
```

Команда del может использоваться для удаления элементов из списка. Вы можете удалить элемент с любым действующим индексом или элемент(-ы) любого допустимого сегмента. Создадим список, а затем воспользуемся del для удаления его последнего элемента:

```
>>> numbers = list(range(0, 10))          # range генерирует числа от 0 до 9
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del numbers[-1]
numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Следующий пример удаляет из списка сегмент из первых двух элементов:

```
>>> del numbers[0:2]
>>> numbers
[2, 3, 4, 5, 6, 7, 8]
```

Метод списков sort *изменяет* список и сортирует элементы по возрастанию:

```
>>> numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Чтобы отсортировать список по убыванию, вызовите sort с необязательным ключевым аргументом reverse, равным True (по умолчанию используется значение False):

```
>>> numbers.sort(reverse=True)
>>> numbers
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Встроенная функция sorted *возвращает новый список*, содержащий отсортированные элементы своей *последовательности*-аргумента — исходная последовательность при этом *не изменяется*.

```
>>> numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
>>> ascending_numbers = sorted(numbers)
>>> ascending_numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers
[10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

Необязательный ключевой аргумент reverse со значением True заставляет функцию отсортировать элементы по убыванию.

Часто бывает нужно определить, содержит ли список (последовательность) значение, соответствующее заданному *ключу поиска*. У списков имеется метод index, которому передается ключ поиска в качестве аргумента. Метод начинает поиск с индекса 0 и возвращает индекс *первого* элемента, который равен ключу поиска:

```
>>> numbers = [3, 7, 1, 4, 2, 8, 5, 6]
>>>  numbers.index(5)                  # 5 – ключ поиска
```

```
6
```

Если искомое значение отсутствует в списке, то происходит ошибка ValueError. Код ниже ищет в списке значение 5, начиная с индекса 7 и до конца списка:

```
>>> numbers = [3, 7, 1, 4, 2, 8, 5, 6, 3, 7, 1, 4, 2, 8, 5, 6]
>>> numbers.index(5, 7)
14
```

Оператор in проверяет, содержит ли итерируемый объект, каковым является список, заданное значение:

```
>>> 1000 in numbers
False
>>> 5 in numbers
True
```

Кортежи

Кортеж (класс tuple) – упорядоченный список значений, отличающийся от списка тем, что может быть использован в качестве ключа в словаре и в качестве элемента множества (см. далее). Обратите внимание, что кортежи заключаются в круглые скобки, а списки - в квадратные скобки.

Чтобы создать пустой кортеж, используйте пустые круглые скобки:

```
>>> student_tuple = ()
>>> student_tuple
()
>>> len(student_tuple)
0
```

Для упаковки элементов в кортеж можно перечислить их, разделяя запятыми:

```
>>> student_tuple = 'John', 'Green', 3.3  #создание кортежа
>>> student_tuple
('John', 'Green', 3.3)
>>> len(student_tuple)
3
>>> student_tupl[0]                       #обращение по индексу
'John'
```

Когда Python выводит кортеж, он всегда отображает его содержимое в круглых скобках. Список значений кортежа, разделяемых запятыми, также можно при создании заключать в круглые скобки (хотя это и не обязательно):

```
>>> pair = (3, 5)          #упаковка значений в кортеж
>>> x, y = pair            #распаковка (извлечение) кортежа
>>> x
3
```

```
>>> y
5
```

Кортеж похож на список, за исключением того, что вы не можете изменить его после создания:

```
>>>pair[1] = 6                #попытка изменения значения элемента кортежа
TypeError: object does not support item assignment
```

Попытка изменить неизменяемую структуру вызывает исключение. Исключения указывают на ошибки, например, выход индекса за границы, ошибки типа и т. п. Тем не менее, можно добавлять элементы в кортеж, кортежи могут содержать изменяемые объекты, кортежи можно добавлять в списки. Пусть

```
>>> tuple1 = (10, 20, 30)
>>> tuple2 = tuple1          # tuple 2 и tulple 1 ссылаются на один и тот же кортеж
>>> tuple2
(10, 20, 30)
```

При конкатенации кортежа (40, 50) с tuple1 создается *новый* кортеж, ссылка на который присваивается переменной tuple1, тогда как tuple2 все еще ссылается на исходный кортеж:

```
# добавление элементов в кортеж
>>> tuple1 += (40, 50)       # справа от += должен быть кортеж
>>> tuple1
(10, 20, 30, 40, 50)
>>> tuple2
(10, 20, 30)
```

Конструкция += также может использоваться для присоединения кортежа к списку:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers += (6, 7)
>>> numbers
[1, 2, 3, 4, 5, 6, 7]
```

Кортежи могут содержать изменяемые объекты. Создадим кортеж student_tuple с именем, фамилией и списком оценок:

```
>>> student_tuple = ('Amanda', 'Blue', [98, 75, 87])
```

И хотя сам кортеж неизменяем, его элемент-список может изменяться:

```
>>>  student_tuple[2][1] = 85
>>> student_tuple
('Amanda', 'Blue', [98, 85, 87])
```

Множества

Множество – это еще одна структура данных, которая является неупорядоченным списком без повторяющихся элементов. Множество можно создать из списка:

```
>>> shapes = ['circle', 'square', 'triangle', 'circle']
>>> setOfShapes = set(shapes)
```

Ниже представлены примеры, как добавлять элементы во множество, проверять, входит ли элемент во множество, и выполнять операции над множествами (разность, пересечение, объединение):

```
>>> setOfShapes
set(['circle', 'square', 'triangle'])
>>> setOfShapes.add('polygon')                    #добавление
>>> setOfShapes
set(['circle', 'square', 'triangle', 'polygon'])
>>> 'circle' in setOfShapes                       #принадлежность
True
>>> 'rhombus' in setOfShapes
False
>>> favoriteShapes = ['circle', 'triangle', 'hexagon']
>>> setOfFavoriteShapes = set(favoriteShapes)
>>> setOfShapes – setOfFavoriteShapes                  #разность
set(['square', 'polygon'])
>>> setOfShapes & setOfFavoriteShapes                  #пересечение
set(['circle', 'triangle'])
>>> setOfShapes | setOfFavoriteShapes                  #объединение
set(['circle', 'square', 'triangle', 'polygon', 'hexagon'])
```

Обратите внимание на то, что множества неупорядочены.

Словари

Словарь (класс dict) – это структура, которая обеспечивает отображение одного объекта (ключ) на другой объект (значение). Словари также называют ассоциативными списками, так как они содержат пары *ключ-значение*, заключаемые в фигурные скобки. Ключ должен быть неизменяемым типом (строка, число или кортеж). Значение может быть любым типом данных Python.

В приведенном ниже примере порядок печати ключей, возвращаемых Python, может отличаться от показанного ниже. Причина в том, что в отличие от списков с фиксированным порядком, словарь - это просто хеш-таблица, для которой нет фиксированного порядка ключей. Порядок следования ключей зависит от того, как

именно алгоритм хеширования сопоставляет ключи с сегментами, и обычно является произвольным. При кодировании Вы не должны полагаться на порядок следования ключей. Примеры основных операций со словарями:

```
>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0}      #создание
>>> studentIds['turing']                                            #поиск по ключу
56.0
>>> studentIds['nash'] = 'ninety-two'                               #замена значения
>>> studentIds
{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}
>>> del studentIds['knuth']                                  #удаление
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds['knuth'] = [42.0, 'forty-two']                       #значение список
>>> studentIds
{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds.keys()                                               #список ключей
['knuth', 'turing', 'nash']
>>> studentIds.values()                                             # список значений
[[42.0, 'forty-two'], 56.0, 'ninety-two']
>>> studentIds.items()                                              #список пар
[('knuth', [42.0, 'forty-two']), ('turing',56.0), ('nash', 'ninety-two')]
>>> len(studentIds)
3
```

Обратите внимание, что метод items возвращает пары *ключ-значение* в форме кортежей.

1.2.5 Написание скриптов

Написание скриптов требует знакомства с управляющими конструкциями Python (if и else, for, while и др.). Так как они подобны управляющим конструкциям других языков программирования, то просто покажем их использование на примерах. Детальнее с ними можно ознакомиться в соответствующем разделе официального руководства Python по адресу: https://docs.python.org/3.6/tutorial/

При написании скриптов будьте внимательны с отступами. В отличие от многих других языков, Python для структурирования программного кода и его корректной интерпретации использует отступы. Так, например, следующий скрипт:

```
if 0 == 1:
    print('We are in a world of sport')
print('Thank you for playing')
```

выведет фразу “Thank you for playing”. Но если мы его перепишем так:

```
if 0 == 1:
    print('We are in a world of sport')
    print('Thank you for playing')
```

то ничего выводиться не будет. Обычно используется отступ в 4 позиции.

Теперь, когда вы научились использовать Python в интерактивном режиме, давайте напишем простой скрипт на Python, демонстрирующий использование цикла for. Откройте файл с именем foreach.py, который содержит следующий код:

```
# цикл по элементам списка fruits
fruits = ['apples', 'oranges', 'pears', 'bananas']
for fruit in fruits:
    print(fruit + ' for sale')

# цикл по парам “ключ-значение”из словаря fruitPrices
fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
for fruit, price in fruitPrices.items():
    if price < 2.00:
        print('%s cost %f a pound' % (fruit, price))
    else:
        print(fruit + ' are too expensive!')
```

Чтобы выполнить скрипт, в командной строке введите команду (предварительно командой cd перейдите в папку с файлом foreach.py):

```
C:\user\user\ai\lab1> python foreach.py
```

В результате выполнения программы получим:

```
apples for sale
oranges for sale
pears for sale
bananas for sale
apples are too expensive!
oranges cost 1.500000 a pound
pears cost 1.750000 a pound
```

Помните, что результаты печати словаря с ценами могут располагаться в другом порядке, чем в показано.

Циклический просмотр элементов списка и их индексов можно организовать с использованием цикла for и функции enumerate:

```
fruits = ['apples', 'oranges', 'pears', 'bananas']
for idx, fruit in enumerate(fruits):   #enumerate возвращает (idx,fruit)
    print(‘#%d:%s’ % (idx, fruit))
```

В результате выполнения этого кода получим:

```
#0:apples
#1:oranges
#2:pears
#3:bananas
```

Часто при создании новых списков используют конструкцию *списковое включение* (list comprehensions), которая обеспечивает компактную замену конструкции с for. Например, следующий код

```
list1 = [ ]
for item in range(1, 6):       # range формирует последовательность от 1 до 5
   list1.append(item)
```

создает список [1, 2, 3, 4, 5]. Этот же список можно создать с помощью спискового включения:

```
list1=[x for x in range(1,6)]
```

Списковое включение позволяет выполнять разные операции (например, вычисления), *отображающие* элементы на новые значения (в том числе, возможно, и других типов). Отображение (mapping) — стандартная операция функционального программирования, которая возвращает результат с *тем же* количеством элементов, что и в исходных отображаемых данных [4, 5].

Другая распространенная операция программирования в функциональном стиле — *фильтрация* элементов и отбор только тех элементов, которые удовлетворяют заданному условию. Как правило, при этом строится список с *меньшим* количеством элементов, чем в фильтруемых данных. Чтобы выполнить эту операцию с использованием спискового включения, используйте *секцию* if. Следующий фрагмент кода демонстрирует простое отображение и фильтрацию с помощью *спискового включения*:

```
#отображение nums на plusOneNums
nums = [1, 2, 3, 4, 5, 6]
plusOneNums = [x + 1 for x in nums]

#фильтрация списков
# создаем список из нечетных элементов nums
oddNums = [x for x in nums if x % 2 == 1]
print(oddNums)

# создаем список из нечетных элементов plusOneNums
oddNumsPlusOne = [x + 1 for x in nums if x % 2 == 1]
print(oddNumsPlusOne)
```

Этот код находится в файле с именем listcomp.py, который вы можете запустить командой:

```
C:\user\user\ai\lab1> python listcomp.py
[1, 3, 5]
[2, 4, 6]
```

1.2.6 Выражения-генераторы

Выражение-генератор отчасти напоминает списковое включение, но оно создает итерируемый *объект-генератор*, производящий значения *по требованию* [5]. Этот механизм называется *отложенным вычислением*. В списковом включении используется *быстрое вычисление,* позволяющее создавать списки в момент выполнения. При большом количестве элементов создание списка может потребовать значительных объёмов памяти и затрат времени. Таким образом, выражения-генераторы могут сократить потребление памяти программой и повысить быстродействие, если не все содержимое списка понадобится одновременно.

Выражения-генераторы обладают теми же возможностями, что и списковое включение, но они определяются в круглых скобках вместо квадратных. Ниже выражение-генератор возвращает квадраты только нечетных чисел из numbers:

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
for value in (x ** 2 for x in numbers if x % 2 != 0):
        print(value, end=' ')
```

В результате выполнения этого кода будут выведены числа: 9 49 1 81 25.

Чтобы показать, что выражение-генератор не создает список, присвоим выражение-генератор из предыдущего фрагмента переменной и выведем значение этой переменной:

```
>>> squares_of_odds = (x ** 2 for x in numbers if x % 2 != 0)
>>> squares_of_odds
<generator object <genexpr> at 0x1085e84c0>
```

Текст "generator object <genexpr>" сообщает, что square_of_odds является объектом-генератором, который был создан на базе выражения-генератора (объект genexpr).

1.2.7 Определение функций

Функция в языке Python определяются с использованием заголовка def *имя функции* (*параметры*). Ниже приведен пример определения и вызова функции, которая выводит стоимость покупки фруктов:

```
fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}

# Определение функции buyFruit
def buyFruit(fruit, numPounds):
    if fruit not in fruitPrices:
        print("Sorry we don't have %s" % (fruit))
    else:
        cost = fruitPrices[fruit] * numPounds
        print("That'll be %f please" % (cost))

# Основная функция
if __name__ == '__main__':
```

```
buyFruit('apples', 2.4)                    # вызов функции buyFruit
buyFruit('coconuts', 2)
```

Проверка __name__ == '__main__' используется для разграничения выражений, которые выполняются, когда файл вызывается как сценарий из командной строки. Код после проверки – это код основной функции. Сохраните этот скрипт как fruit.py и запустите его:

```
C:\user\user\ai\lab1> python fruit.py
That'll be 4.800000 please
Sorry we don't have coconuts
```

1.2.7. Функционалы: filter и map. Лямбда выражения.

Ранее были представлены некоторые средства программирования в функциональном стиле — применение спискового включения для фильтрации и отображения. Здесь продемонстрируем применение встроенных функций filter и map для фильтрации и отображения, соответственно.

Иногда необходимо передать в функцию через ее формальный параметр имя другой функции. Такой параметр называют *функциональным*, а функцию, принимающую этот параметр — *функционалом*. Функция также может возвращать в виде результата другую функцию. Такие функции называют функциями *с функциональным значением*. Вызов функции с функциональным значением может быть аргументом функционала, а также использоваться вместо имени функции в вызове. Функционалы и функции с функциональным значением относятся к инструментарию функционального программирования и называются функциями *высших порядков*.

Первым аргументом filter должна быть некоторая функция-предикат, которая получает один аргумент и возвращает True, если значение должно включаться в результат. Например, такой функцией-предикатом может быть is_odd(x):

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
def is_odd(x):
    """Возвращает True только для нечетных x."""
    return x % 2 != 0
```

Теперь воспользуемся встроенной функцией filter для получения нечетных значений списка numbers. Вызов list(filter(is_odd, numbers)) вернет список [3, 7, 1, 9, 5]. Здесь функция is_odd возвращает True, если ее аргумент является нечетным. Функция filter вызывает is_odd по одному разу для каждого значения numbers. Функция filter возвращает итератор, так что для получения результатов filter нужно будет выполнить их перебор. Это еще один пример отложенного вычисления. Функция list перебирает результаты и создает список, в котором они содержатся.

Для простых функций (типа is_odd), возвращающих только *значение одного выражения*, можно использовать *лямбда-выражение* для определения встроенной функции в том месте, где она нужна, — обычно при передаче другой функции в

качестве параметра, например: list(filter(lambda x: x % 2 != 0, numbers)). Лямбда-выражение является *анонимной функцией*, то есть функцией, не имеющей имени. В вызове filter(lambda x: x % 2 != 0, numbers) первым аргументом является лямбда-выражение

```
lambda x: x % 2 != 0
```

После lambda следует разделяемый запятыми список параметров, двоеточие (:) и выражение. В данном случае список параметров состоит из одного параметра с именем x. Лямбда-выражение *неявно* возвращает значение своего выражения. Таким образом, любая простая функция в форме

```
def имя_функции(список_параметров):
    return выражение
```

может быть выражена в более компактной форме посредством лямбда-выражения

```
lambda список_параметров: выражение
```

Воспользуемся встроенной функцией map с лямбда-выражением для возведения в квадрат каждого значения из numbers:

```
>>> numbers=[10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
>>> list(map(lambda x: x ** 2, numbers))
[100, 9, 49, 1, 81, 16, 4, 64, 25, 36]
```

Первым аргументом функции map является функция, которая получает одно значение и возвращает новое значение — в данном случае лямбда-выражение, которое возводит свой аргумент в квадрат. Вторым аргументом является итерируемый объект с отображаемыми значениями. Функция map использует отложенное вычисление, поэтому возвращаемый map итератор передается функции list. Это позволит перебрать и создать список отображенных значений.

Предшествующие операции filter и map можно объединить следующим образом:

```
>>>list(map(lambda x: x ** 2,  filter(lambda x: x % 2 != 0, numbers)))
[9, 49, 1, 81, 25]
```

1.2.8. Классы и объекты Python

Класс инкапсулирует данные и функции для взаимодействия с этими данными. Пример определения класса FruitShop (магазин фруктов, класс определен в файле shop.py):

```
class FruitShop:

    def __init__(self, name, fruitPrices):            # конструктор класса
        """
```

```
            name: Название магазина фруктов
            fruitPrices: Словарь с ключами в виде названий фруктов
            и ценами в виде значений, например:
            {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
        """

        self.fruitPrices = fruitPrices
        self.name = name
        print('Welcome to %s fruit shop' % (name))

    def getCostPerPound(self, fruit):
        """
            Возвращает стоимость фрукта 'fruit''
            если он в словаре, иначе - None
            fruit:строка с названием фрукта
        """
        if fruit not in self.fruitPrices:
            return None
        return self.fruitPrices[fruit]

    def getPriceOfOrder(self, orderList):
        """
            Возвращает стоимость заказа orderList, включая только
            фрукты, которые есть в магазине.
            orderList: Список из кортежей (fruit, numPounds)
        """
        totalCost = 0.0
        for fruit, numPounds in orderList:
            costPerPound = self.getCostPerPound(fruit)
            if costPerPound != None:
                totalCost += numPounds * costPerPound
        return totalCost

    def getName(self):
        return self.name
```

Класс FruitShop содержит название магазина и цены за фунт некоторых фруктов, а также предоставляет функции или методы для этих данных. В чем преимущество обертывания этих данных в класс?

- инкапсуляция данных предотвращает их изменение или ненадлежащее использование;

- абстракция, которую предоставляют классы, упрощает написание кода общего назначения.

Использование объектов

Как создать объект (экземпляр класса) и использовать его? Убедитесь, что у вас есть реализация FruitShop в shop.py. Затем импортируете код из этого файла (делая его доступным для других скриптов) с помощью import shop. Затем создайте объект типа FruitShop следующим образом:

```
import shop

# создание объекта-магазина 1 и его обработка
shopName = 'the Berkeley Bowl'
fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
berkeleyShop = shop.FruitShop(shopName, fruitPrices)
applePrice = berkeleyShop.getCostPerPound('apples')
print(applePrice)
print('Apples cost $%.2f at %s.' % (applePrice, shopName))

# создание объекта- магазина  2 и его обработка
otherName = 'the Stanford Mall'
otherFruitPrices = {'kiwis': 6.00, 'apples': 4.50, 'peaches': 8.75}
otherFruitShop = shop.FruitShop(otherName, otherFruitPrices)
otherPrice = otherFruitShop.getCostPerPound('apples')
print(otherPrice)
print('Apples cost $%.2f at %s.' % (otherPrice, otherName))
print("That's expensive!")
```

Этот код находится в файле shopTest.py, запустить его можно так:

```
C:\user\user\ai\lab1>  python shopTest.py
Welcome to the Berkeley Bowl fruit shop
1.0
Apples cost $1.00 at the Berkeley Bowl.
Welcome to the Stanford Mall fruit shop
4.5
Apples cost $4.50 at the Stanford Mall.
That's expensive!
```

Проанализируем код. Оператор import shop обеспечивает загрузку определения класса из shop.py. Строка berkeleyShop = shop.FruitShop(shopName, fruitPrices) создает экземпляр (объект) класса FruitShop, определенного в shop.py, путем вызова конструктора __init__ этого класса. Обратите внимание, что мы передали только два аргумента, в то время как __init__ принимает три аргумента: (self, name, fruitPrices). Причина этого в том, что все методы в классе имеют в качестве первого аргумента self. Значение переменной self автоматически присваивается самому объекту; при вызове метода вы предоставляете только оставшиеся аргументы. Переменная self содержит все данные (name и fruitPrices) для текущего конкретного экземпляра.

Статические переменные класса и переменные экземпляра класса

В следующем примере показано, как использовать статические переменные класса и переменные экземпляра класса в Python. Создайте файл person_class.py, содержащий следующий код:

```
class Person:
```

```
    population = 0

    def __init__(self, myAge):
        self.age = myAge
        Person.population += 1

    def get_population(self):
        return Person.population

    def get_age(self):
        return self.age
```

Выполним скрипт:

```
C:\user\user\ai\lab1> python person_class.py
```

Используем класс, следующим образом:

```
>>> import person_class
>>> p1 = person_class.Person(12)
>>> p1.get_population()
1
>>> p2 = person_class.Person(63)
>>> p1.get_population()
2
>>> p2.get_population()
2
>>> p1.get_age()
12
>>> p2.get_age()
63
```

В приведенном выше коде age (возраст) – это переменная экземпляра класса, а population (население) – статическая переменная класса. Статическая переменная population используется всеми экземплярами класса Person, тогда как каждый экземпляр имеет свою собственную переменную age.

1.2.9. Дополнительные советы и хитрости Python

Рассмотрим несколько полезных советов.

1. Применяйте функцию range для генерации последовательности целых чисел, используемых в качестве значений переменной цикла for. Последовательность справа от ключевого слова in в команде for должна быть *итерируемым объектом*, то есть объектом, из которого команда for может брать элементы по одному, пока не будет обработан последний элемент. Итератор напоминает закладку в книге — он всегда знает текущую позицию последовательности, чтобы вернуть следующий элемент по требованию.

Рассмотрим цикл for и встроенную *функцию* range для выполнения ровно 10 итераций с выводом значений от 0 до 9:

```
>>> for counter in range(10):
        print(counter, end=' ')

0 1 2 3 4 5 6 7 8 9
```

Вызов функции range(10) создает итерируемый объект, который представляет последовательность целых чисел 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Команда for прекращает работу при завершении обработки последнего целого числа, выданного range.

2. После импорта файла, если вы редактируете исходный файл, изменения не будут немедленно переданы в интерпретатор. Для этого используйте команду перезагрузки:

```
>>> reload(shop)
```

3. Рассмотрим некоторые проблемы, с которыми обычно сталкиваются изучающие Python.

Проблема: ImportError: нет модуля с именем py

Решение: для операторов импорта import <*имя-пакета*> не включайте в имя пакета расширение файла (т. е. подстроку .py).

Проблема: NameError: имя «MY_VAR» не определено. Даже после выполненного импорта вы можете увидеть такое сообщение.

Решение: чтобы получить доступ к элементу модуля, вы должны использовать module_name.member_name, где module_name - это имя файла, а member_name - имя переменной (или функции), к которой вы пытаетесь получить доступ.

Проблема: TypeError: объект dict не может быть вызван.

Решение: поиск элементов в словаре выполняется с использованием квадратных скобок [ ], а не круглых скобок () .

Проблема: ValueError: слишком много значений для распаковки.

Решение: убедитесь, что количество переменных, назначаемых в цикле for, совпадает с количеством элементов в каждом элементе списка. Аналогично при работе с кортежами. Например, если пара является кортежем из двух элементов (например, pair = ('apple', 2.0)), то следующий код вызовет ошибку “not enough values to unpack ”:

```
(a, b, c) = pair
```

Вот проблемный сценарий, связанный с циклом for:

```
pairList = [('apples', 2.00), ('oranges', 1.50), ('pears', 1.75)]
for fruit, price, color in pairList:
    print('%s fruit costs %f and is the color %s' % (fruit, price, color))
```

```
ValueError: not enough values to unpack (expected 3, got 2)
```

1.3. Задания для выполнения

Задание 1. Строки

Используя команды dir и help, изучите следующие методы строкового типа: ‘format’, ‘strip’, ‘lstrip’, ‘rstrip’, ‘capitalize’, ‘title’, ‘count’, ‘index’, ‘rindex’, ‘startswith’, ‘endswith’, ‘replace’, ‘split’, ‘rsplit’, ‘join’, ‘partition’, ‘rpartition’. Разработайте примеры вызова указанных методов и внесите результаты в отчет.

Задание 2. Списки

Используя команды dir и help, изучите, изучите следующие методы обработки списков: 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'. Разработайте примеры вызова указанных методов и внесите результаты в отчет.

Задание 3. Словари

Используя команды dir и help, изучите, изучите следующие методы обработки словарей: 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', setdefault', 'update', 'values'. Разработайте примеры вызова указанных методов и внесите результаты в отчет.

Задание 4. Списковое включение

Определите списковое включение, которое из списка строк генерирует версию нового списка, состоящего из строк, длина которых больше пяти и которые записаны символами нижнего регистра. Решение можно проверить, просмотрев файл listcomp2.py.

Задание 5. Быстрая сортировка

Напишите функцию быстрой сортировки на Python, используя списки. Используйте первый элемент как точку деления списка. Вы можете проверить своё решение, просмотрев файл quickSort.py.

Задание 6. Решение задач и автооценивание

Чтобы ознакомиться с автооцениванием, мы попросим вас написать код, протестировать и включить в отчет решения **для трех задач, приведенных ниже**. Загрузите архив (codingdiagnostic_r.zip) с системой автооценивания в **отдельную рабочую папку**. Разархивируйте его и изучите содержимое. Архив содержит ряд файлов, которые вы будете *редактировать или запускать*:

addition.py: исходный файл для задачи (вопроса) 1
buyLotsOfFruit.py: исходный файл для задачи (вопроса) 2
shop.py: исходный файл для задачи (вопроса) 3
shopSmart.py: исходный файл для задачи (вопроса) 3
autograder.py: скрипт автооценивания (см. ниже)

Другие файлы в архиве можно *игнорировать*:

test_cases: каталог содержит тестовые примеры для каждой задачи (вопроса)
grading.py: код автооценивателя
testClasses.py: код автооценивателя
tutorialTestClasses.py: тестовые классы для этого лабораторного задания
projectParams.py: параметры

Команда python autograder.py оценит ваше решение для всех трех задач (вопросов). Если запустить автооцениватель перед редактированием каких-либо файлов, то получим ответ, фрагмент которого приведен ниже:

```
>>>  python autograder.py
Starting on 8-20 at 19:33:19

Question q1
===========

*** FAIL: test_cases\q1\addition1.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "2"
*** FAIL: test_cases\q1\addition2.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "5"
*** FAIL: test_cases\q1\addition3.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "7.9"
*** Tests failed.

### Question q1: 0/1 ###

Question q2
===========

*** FAIL: test_cases\q2\food_price1.test
,,,,

### Question q2: 0/1 ###

Question q3
===========

Welcome to shop1 fruit shop
…
***      correct result: "<FruitShop: shop3>"
```

```
*** Tests failed.

### Question q3: 0/1 ###

Finished at 19:33:19

Provisional grades
==================
Question q1: 0/1
Question q2: 0/1
Question q3: 0/1
------------------
Total: 0/3

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

Для каждой из трех задач (вопросов) отображены результаты тестов, оценка и окончательное резюме в конце. Поскольку задачи еще не решены, все тесты не пройденны. По мере решения каждой задачи вы можете обнаружить, что некоторые тесты проходят успешно, а другие - нет. Когда все тесты будут пройдены, вы получаете итоговую оценку. Глядя на результаты для задачи (вопроса 1), вы можете увидеть, что не пройдены три теста с сообщением об ошибке «add (a, b) must return the sum of a and b». Ваш код всегда дает результат 0, но правильный ответ имеет другое значение. Это можно исправить следующим образом.

Задача 1. Функция сложения

Откройте addition.py и посмотрите шаблон определения функции add:

```
def add(a, b):
    "Возвращает сумму a и b"
    "*** ВСТАВЬТЕ ВАШ КОД СЮДА***"
    return 0
```

Тесты вызвали эту функцию с разными значениями a и b, но код всегда возвращал ноль. Измените это определение на следующее:

```
def add(a, b):
    ""Возвращает сумму a и b"
    print("Passed a = %s and b = %s, returning a + b = %s" % (a, b, a + b))
    return a + b
```

Теперь перезапустите автооцениватель и получите следующий результат (без результатов тестов для вопросов 2 и 3):

```
>>> python autograder.py -q q1
Starting on 8-20 at 19:47:54

Question q1
===========
```

```
Passed a = 1 and b = 1, returning a + b = 2
*** PASS: test_cases\q1\addition1.test
***      add(a,b) returns the sum of a and b
Passed a = 2 and b = 3, returning a + b = 5
*** PASS: test_cases\q1\addition2.test
***      add(a,b) returns the sum of a and b
Passed a = 10 and b = -2.1, returning a + b = 7.9
*** PASS: test_cases\q1\addition3.test
***      add(a,b) returns the sum of a and b

### Question q1: 1/1 ###
…
Finished at 19:47:54

Provisional grades
==================
Question q1: 1/1
Question q2: 0/1
Question q3: 0/1
------------------
Total: 1/3
```

Теперь вы прошли все тесты и получили итоговую оценку за решение задачи 1. Обратите внимание на новые строки «Passed a =…», которые появляются перед «*** PASS:…». Они создаются оператором печати в функции add. Вы можете использовать подобные операторы печати для вывода информации, полезной для отладки.

Задача 2. Функция buyLotsOfFruit

Определите функцию buyLotsOfFruit(orderList) в файле buyLotsOfFruit.py, которая принимает список-заказ orderList, состоящий из кортежей (фрукт, вес), например,

```
orderList = [('apples', 2.0), ('pears', 3.0), ('lime', 4.0)]
```

и возвращает стоимость заказа. Для вычисления стоимости заказа функция использует ценник

```
fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75, 'limes': 0.75, 'strawberries': 1.00}
```

Если в списке-заказе есть название фрукта, которого нет в ценнике fruitPrices, то функция должна вывести сообщение об ошибке и вернуть None. Пожалуйста, не изменяйте переменную fruitPrices. Запускайте python autograder.py, пока не пройдут все тесты для задачи 2. Каждый тест подтверждает, что buyLotsOfFruit(orderList) возвращает правильный ответ с учетом различных возможных входных данных.

Например, test_cases / q2 / food_price1.test проверяет, равна ли стоимость списка заказа [('apples', 2.0), ('pears', 3.0), ('lime', 4.0)] значению 12,25

Задача 3. Функция shopSmart

Определите функцию shopSmart(orderList, fruitShops) в файле shopSmart.py, которая принимает список заказов orderList и список магазинов FruitShops, и возвращает название магазина, где ваш заказ будет иметь наименьшую стоимость. Пожалуйста, не меняйте имя файла или имена переменных. Для решения задачи используйте класса FruitShop в файле shop.py (см. описание класса выше в п.1.2.8). Запускайте python autograder.py, пока не будут пройдены все тесты.

Каждый тест подтверждает, что shopSmart(orders, shops) возвращает правильный ответ с учетом различных возможных исходных данных. Например, при следующих значениях переменных

```
orders1= [('apples', 1.0), ('oranges', 3.0)]
orders2 = [('apples', 3.0)]
dir1 = {'apples': 2.0, 'oranges': 1.0}
shop1 = shop.FruitShop('shop1', dir1)
dir2 = {'apples': 1.0, 'oranges': 5.0}
shop2 = shop.FruitShop('shop2', dir2)
shops = [shop1, shop2]
```

Тест test_cases/q3/select_shop1.test проверяет, что shopSmart(orders1,shops) == shop1, а тест test_cases/q3/select_shop2.test проверяет, действительно ли shopSmart(orders2 ,shops) == shop2.

После завершения отладки скопируйте результаты автооценивания в отчет.

1.4. Порядок выполнения лабораторной работы

1.4.1. Изучите по материалам раздела 1.2 и [5] основы языка Python, структуры данных и методы обработки списков, кортежей, множеств, словарей, классы и объекты Python, среду программирования на языке Python. Проверьте выполнение приведенных примеров в среде программирования.

1.4.2. Выполните задания 1 – 5 из раздела 1.3 в интерактивном режиме, используя возможности IPython. Результаты выполнения каждого задания внесите в отчет.

1.4.3. Определите в соответствии с заданием 6 из раздела 1.3 функции и методы для решения 3-х сформулированных задач. Используйте для редактирования функций и выполнения кода интегрированную среду Spyder IDE. Зафиксируйте результаты выполнения функций во всех необходимых режимах. Выполните с помощью autograder.py автооценивание. При обнаружении ошибок отредактируйте код. Результаты автооценивания внесите в отчет.

1.5. Содержание отчета

Цель работы, задания 1-5 с примерами выполнения, описание класса для задания 6 и определений собственных функций, результаты выполнения всех собственных функций задания 6, результаты автооценивания задания 6, выводы.

1.6. Контрольные вопросы

1.6.1. Какие типы числовых данных поддерживает Python? Как определить тип переменной в Python?

1.6.2. Как выполнить целочисленное деление и вычисление остатка от деления целых чисел?

1.6.3. Приведите примеры использования логических операций и операций отношений.

1.6.4. Как выполнить конкатенацию строк, заменить символы нижнего регистра на верхний и наоборот, определить длину строки?

1.6.5. Приведите пример строки форматированного вывода в операторе print, объясните спецификации форматирования и назначение управляющих последовательностей.

1.6.6. Что такое f-стока? Приведите примеры использования.

1.6.7. Объясните назначение методов для работы со строками: 'format', ' 'index', 'isalpha', 'isdigit', 'partition', 'replace', 'split', 'title', 'translate', 'upper', 'zfill'.

1.6.8. Как создать список? Как выполняется обращение к элементам списков, сегментам списков? Объясните назначение методов для работы со списками: pop, append, insert, sort, index.

1.6.9. Как создать кортеж? Как его распаковать? Как обратиться к элементам кортежа? Как объединить список и кортеж?

1.6.10. Как создать множество? Как проверить принадлежность элемента множеству? Приведите примеры операций с множествами.

1.6.11. Что такое словарь? Как его создать? Как выполнить поиск в словаре? Как заменить значение в словаре? Как удалить элемент словаря? Как получить список ключей, список значений, список кортежей ключ-значение?

1.6.12. Напишите скрипт, печатающий номера элементов списка и сами элементы.

1.6.13.Что такое списковое включение? Приведите примеры операций отображения и фильтрации с помощью спискового включения.

1.6.14. Что такое выражение-генератор? Приведите пример применения в цикле for.

1.6.15. Как определить собственную функцию в Python? Определите функцию быстрой сортировки.

1.6.16. Что такое функционал? Как определяется анонимная функция? Приведите примеры использования функционалов filter и map.

1.6.17. Как определяется класс в языке Python? Приведите пример определения класса. Что такое статическая переменная класса? Чем она отличается от переменной экземпляра класса?

2. ЛАБОРАТОРНАЯ РАБОТА № 2 «ИССЛЕДОВАНИЕ НЕИНФОРМИРОВАННЫХ МЕТОДОВ ПОИСКА РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»

2.1. Цель работы

Исследование неинформированных методов поиска решений задач в пространстве состояний, приобретение навыков программирования интеллектуальных агентов, планирующих действия на основе методов слепого поиска решений задач.

2.2. Краткие теоретические сведения

2.2.1. Обучающая среда AI Pacman

Обучающая среда AI Pacman была разработана Джоном ДеНеро и Дэном Кляйном [6] для обучения искусственному интеллекту. Оригинальная видео игра ***Pac-Man*** была разработана японской компанией Namco в1980 году.

Игра Pacman проста: Pacman (герой игры) должен пройти лабиринт (рисунок 2.1) и съесть все (маленькие) гранулы, не будучи съеденным злобными привидениями, которые охотятся на него. Если Pacman съест одну из (больших) энергетических гранул, он становится невосприимчивым к привидениям на определенный период времени и получает способность поедать призраков, зарабатывая очки.

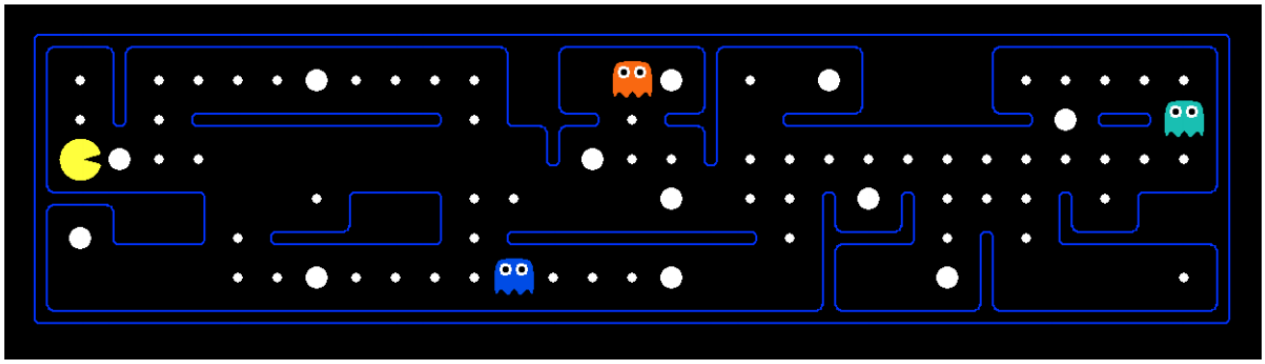


Рисунок 2.1 – Экран обучающей среды AI Pacman

2.2.2. Агенты

Основной задачей искусственного интеллекта является создание **рационального агента** – сущности, которая имеет цели, воспринимает среду и пытается выполнить серию **действий**, которые приведут к наилучшему/оптимальному ожидаемому результату с учетом этих целей. Рациональные агенты существуют в **среде**, специфичной для заданного агента. Например, для шашечного агента среда – это виртуальная шашечная доска, на которой он играет против оппонентов, где ходы фигур – это действия. Вместе среда и агенты создают некоторый **мир**.

Рефлекторный агент – это агент, который не оценивает последствия своих действий, и, скорее, выбирает действие, основываясь исключительно на текущем состоянии мира. Такие агенты обычно проигрывают агентам, **планирующим** свои

действия, которые поддерживают некоторую модель мира и используют её для симуляции выполнения различных действий. Кроме этого, агент может строить гипотезы о предполагаемых последствиях действий и выбирать лучшие из них. Таким образом, агент моделирует одну из интеллектуальных функций – продумывание действий наперед.

2.2.3. Формулировка задач поиска в пространстве состояний

Чтобы создать рационального агента, планирующего действия, нужен способ математического описания среды, в которой функционирует агент. Для этого мы должны формально представить задачу поиска, учитывая текущее состояние агента и то, как агент может перейти в новое состояние, которое наилучшим образом удовлетворяет цели функционирования агента. Представление задачи в такой постановке соответствует поиску в пространстве состояний и определяется совокупностью четырех составляющих

$$(S_0, S, F, G), \qquad (2.1)$$

где S — множество (пространство) состояний задачи; S_0 — множество начальных состояний, $S_0 \in S$; F — множество функций, преобразующих одни состояния в другие; G — множество целевых состояний, $G \in S$ [1].

Каждая функция $f \in F$, отображает одно состояние в другое — $s_j = f(s_i)$, где $s_i, s_j \in S$. Решением задачи является последовательность функций (действий) f_i, преобразующих начальные состояния в конечные, т.е. $f_n\ f_{n-1}(\ldots(f_2(f_1(S_0)))\ldots)) \in G$. Если такая последовательность не одна и задан критерий оптимальности, то возможен поиск оптимальной последовательности.

Поиск решения в пространстве состояний представляют в виде ***графа состояний.*** Множество вершин графа соответствует состояниям задачи, а множество дуг (ребер) — операторам. Тогда решение задачи может интерпретироваться как поиск пути на графе из исходного состояния задачи в целевое состояние.

Формулировка задачи поиска в пространстве состояний требует определения всех составляющих выражения (2.1):

- пространство состояний (state space) — множество всех состояний, которые возможны мире агента;

- начальное состояние (start state) — состояние, в котором агент существует изначально;

- функция-преемника (**successor function**) — функция, которая принимает на вход состояние и действие, вычисляет стоимость выполнения этого действия и определяет состояние-преемник (**successor**), в котором находился бы агент, если бы выполнил это действие.

- целевогой тест — функция (goal test), которая принимает на вход состояние и определяет, является ли оно целевым состоянием.

По сути, проблема поиска решается следующим образом: сначала рассматривается начальное состояние, затем исследуется пространство состояний с помощью

функции-преемника, которая итеративно вычисляет преемников различных состояний, пока мы не достигнем целевого состояния, после чего определяется путь из начального состояния в целевое состояние (обычно его называется планом).

Процесс применения **функций-приемников** к некоторой вершине графа с целью получения всех ее дочерних вершин (приемников) называется ***раскрытием вершины*.**

2.2.4. Формулировки задач поиска в среде AI Pacman

Рассмотрим вариант игры, в которой имеется лабиринт, Pacman и пищевые гранулы. В этом случае можно сформулировать две отдельные задачи поиска: нахождение пути в лабиринте и поедание всех гранул. В первом случае мы будем пытаться решить проблему оптимального перехода из позиции (x1; y1) в позицию (x2; y2) в лабиринте, в то время как при решении второй задачи наша цель будет заключаться в поедании всех пищевых гранул в кратчайшие сроки. Ниже для этих задач определяются: состояния, действия, функция-преемник и целевой тест.

Нахождение пути:	**Поедание всех гранул**:
- *состояния*: координаты (x, y) местоположений; - *действия*: Север, Юг, Восток, Запад; - *преемник*: обновить только местоположение; - *проверка цели*: (x, y) = END?	- *состояния*: координаты (x, y) местоположений, логические значения точек расположения гранул (true/false); - *действия*: Север, Юг, Восток, Запад; - *преемник*: обновить местоположение и логические значения точек; - *проверка цели*: все ли логические значения точек ложны?

Обратите внимание, что для задачи нахождения пути состояния содержат меньше информации, чем состояния для задачи поедания всех гранул. Так как во втором случае мы должны дополнительно поддерживать массив логических значений, соответствующих каждой грануле, независимо от того, была ли она съедена в данном состоянии или нет. **Состояние мира (world state) агента** может содержать дополнительную информацию, например, информацию о расстоянии, пройденным агентом, или информацию о всех позициях, посещенных агентом, кроме текущего (x, y) местоположения агента и логических значений точек.

2.2.5. Методы поиска решений задач в пространстве состояний

Методы поиска в пространстве состояний подразделяются на две группы: методы неинформированного (слепого) и информированного (упорядоченного, эвристического) поиска. В методах “слепого” поиска выполняется полный просмотр всего пространства состояний, что может приводить к проблеме *комбинаторного взрыва*. К методам слепого поиска относят: поиск в ширину, поиск на основе алгоритма равных цен, различные виды поиска в глубину. Стратегии поиска в пространстве состояний обычно сравнивают с помощью критериев, указанных ниже [4].

Полнота. Гарантируется ли обнаружение решения, если оно существует?

Оптимальность. Стратегию называют оптимальной, если она обеспечивает нахождение решения, которое не обязательно будет наилучшим, но известно, что оно принадлежит подмножеству решений, обладающих некоторыми заданными свойствами, согласно которым мы относим их к оптимальным.

Минимальность. Стратегию называют минимальной, если она гарантирует нахождение наилучшего решения, т.е. минимальность является более сильным случаем оптимальности.

Временная сложность. Время (число операций), необходимое для нахождения решения.

Пространственная сложность. Объем памяти, необходимый для решения задачи.

Рассматриваемые ниже методы поиска используют два списка: список открытых вершин и список закрытых вершин. В *списке открытых вершин* **OPEN** (часто в программах такой список именуют как **FRINGE** или **FRONTIER**) находятся вершины, подлежащие раскрытию; *в списке закрытых вершин* (**CLOSED**) находятся уже раскрытые вершины. Список **CLOSED** позволяет запоминать рассмотренные вершины с целью исключения их повторного раскрытия.

2.2.6. Методы неинформированного (слепого) поиска

Поиск в ширину

Рассмотрим алгоритм поиска в ширину (**BFS- Breadth First Search).** В начале поиска список **CLOSED** пустой, а **OPEN** содержит только начальную вершину. На каждой итерации из списка **OPEN** выбирается для раскрытия первая вершина. Если эта вершина не целевая, то она перемещается в список **CLOSED**, а ее дочерние вершины помещаются в ***конец*** списка **OPEN**, т.е. принцип формирования списка **OPEN** соответствует ***очереди***. Далее процесс повторяется.

Согласно этой стратегии, при построении дерева поиска вершины с глубиной ***k*** раскрываются только после того, как будут раскрыты все вершины предыдущего уровня ***k-1***. В этом случае фронт поиска (fringe, frontier) растет в ширину. Для построения обратного пути (из целевой вершины в начальную вершину) все дочерние вершины снабжаются указателями на соответствующие родительские вершины.

Ниже приведен псевдокод алгоритма поиска в ширину на графе состояний. Функция **pop()** извлекает из списка (в данном случае очереди) **OPEN** первую вершину, функция **push()** выполняет вставку вершины в конец очереди. Вызов функции **problem.getStartState()** обеспечивает получение стартовой вершины (состояния) решаемой задачи **problem**. Параметр **problem** содержит компьютерное представление описания задачи в соответствии с формулой (2.1).

```
def  breadthFirstSearch (problem):
   Определить стартовую вершину: start = problem.getStartState()
   Поместить стартовую вершину в список OPEN: OPEN.push(start)
   CLOSED = [ ]
   Путь = [ ]
```

```
while not OPEN.isEmpty() :
        node = OPEN.pop()
        If node == 'целевая вершина': return Путь
        CLOSED = CLOSED.append(node)
        Раскрыть node и поместить все дочерние вершины, отсутствующие в
        списке CLOSED или OPEN, в конец списка OPEN, связав с каждой до-
        черней вершиной указатель на node
return 'НЕУДАЧА'
```

Если целевая вершина найдена, то алгоритм должен вернуть **Путь** от стартовой вершины до целевой. Значение переменной **Путь** может быть определено на основе дополнительного анализа списка **CLOSED** либо, в простых случаях, путь к каждой вершине накапливается и хранится в самой вершине.

Если повторяющиеся вершины не исключаются из рассмотрения (т.е. не выполняется проверка принадлежности дочерних вершин списку **CLOSED** или **OPEN**), то алгоритм строит не граф поиска, а **дерево поиска**! В этом случае можно легко оценить временную и пространственную сложности алгоритма.

Если каждая вершина дерева поиска имеет B дочерних вершин, то при остановке поиска на глубине, равной d, максимальное число раскрытых вершин будет равно $B+B^2+B^3+\ldots+B^d+(B^{d+1}-B)$ [4]. Обычно вместо этой формулы употребляют её обозначение $O(B^{d+1})$, которое называют ***экспоненциальной оценкой сложности***. Если полагать, что раскрытие каждой дочерней вершины требует одной единицы времени, то $O(B^{d+1})$ является ***оценкой временной сложности***. При этом каждая вершина должна сохраняться в памяти до получения решения. Поэтому ***оценка пространственной сложности*** будет также равна $O(B^{d+1})$. Это приводит к тому, что стратегия поиска в ширину может использоваться только для задач, которые характеризуются пространством поиска небольшой размерности. Но при этом она удовлетворяет ***критерию полноты и минимальности*** (обеспечивает нахождение целевого состояния, которое находится на минимальной глубине дерева поиска, т.е. поиск в ширину обеспечивает нахождение ***самого «поверхностного» решения***).

Поиск по критерию стоимости (алгоритм равных цен)

Поиск в ширину является оптимальным, если стоимости раскрытия всех вершин равны. Если с каждой вершиной связать стоимость её раскрытия и на каждом шаге из списка **OPEN** выбирать для ракрытия вершину с наименьшей стоимостью, то рассмотренная выше процедура будет обеспечивать нахождение **оптимального** решения по критерию стоимости. Стоимость раскрытия $g(V_i)$ некоторой вершины V_i равна

$$g(V_i)=g(V)+c(V,V_i), \qquad (2.2)$$

где V — родительская вершина для вершины V_i; $c(V,V_i)$ — стоимость пути из вершины V в вершину V_i; $g(V)$ — стоимость раскрытия родительской вершины (для начальной вершины $g(V)=0$).

Рассмотренная стратегия гарантирует **полноту** поиска, если стоимость каждого участка пути положительная величина. Так как поиск в этом случае направляется стоимостью путей, то ***временная и пространственная сложности*** (при построении дерева поиска) в наихудшем случае будут пропорциональны $O(B^{1+C/c})$, где C — стоимость оптимального решения, c — средняя минимальная стоимость действия. Эта оценка может быть больше $O(B^d)$. Это связано с тем, что процедура поиска по критерию стоимости часто обследует поддеревья поиска, состоящие из мелких участков небольшой стоимости, прежде чем перейти к исследованию путей, в которые входят крупные, но возможно более полезные участки [4].

Поиск в глубину

При поиске в глубину всегда раскрывается самая глубокая вершина из текущего фронта поиска. Процедура поиска в глубину отличается от процедуры поиска в ширину тем, что дочерние вершины, получаемые при раскрытии вершины **node**, помещаются в **начало** списка **OPEN,** т.е. принцип формирования списка **OPEN** соответствует **стеку**.

Поиск в глубину требует хранения только одного пути от корня до листового узла. Для дерева поиска с коэффициентом ветвления B и максимальной глубиной m поиск в глубину требует хранения $Bm+1$ узлов, т.е ***пространственная сложность*** соответствует $O(Bm)$, что намного меньше по сравнению с рассмотренными выше стратегиями поиска [4]. При **поиске в глубину с возвратами** потребуется еще меньше памяти. В этом случае каждый раз формируется только одна из дочерних вершин и запоминается информация о том, какая вершина должна быть сформирована следующей. Таким образом, требуется только $O(m)$ ячеек памяти.

Однако поиск в глубину ***не является полным*** (в случае не ограниченной глубины) и ***оптимальным*** (не обеспечивает гарантированное нахождение наиболее поверхностного целевого узла). В наихудшем случае ***временная сложность*** пропорциональна $O(B^m)$, где m — максимальная глубина залегания решения (m может быть гораздо больше по сравнению с d — глубиной самого поверхностного решения).

Проблему деревьев поиска неограниченной глубины можно решить, предусматривая применение во время поиска заранее определенного ***предела глубины L.*** Это означает, что вершины на глубине L рассматриваются таким образом, как если бы они не имели дочерних вершин. Такая стратегия поиска называется **поиском с ограничением глубины**. Однако при этом вводится дополнительный источник **неполноты**, если будет выбрано $L<d$, т.е. самая поверхностная цель находится за пределами глубины. А при выборе $L>d$ поиск с ограничением глубины будет **неоптимальным** (не гарантируется получение самого поверхностного решения).

Поиск в глубину с итеративным углублением. Эта стратегия поиска позволяет найти наилучший предел глубины. Для этого применяется процедура поиска с ограничением по глубине. При этом предел глубины постепенно увеличивается (в начале он равен 0, затем 1, затем 2 и т.д.) до тех пор пока не будет найдена цель. Это происходит, когда предел глубины достигает значения d — глубины са-

мого поверхностного решения. Конечно, здесь допускается повторное обследование одних и тех же состояний. Однако такие повторные операции не являются слишком дорогостоящими, и временная сложность оценивается значением $O(B^d)$ [3].

В поиске с итеративным углублением (по дереву поиска) сочетаются преимущества поиска в ширину (является **полным**) и поиска в глубину (малое значение **пространственной сложности**, равное $O(Bd)$).

2.2.7 Общие сведения о программировании в среде AI Pacman

Код среды AI Pacman, используемый в данной лабораторной работе, находится в нескольких файлах. Некоторые из файлов необходимо будет прочитать и понять, чтобы выполнить задание, а некоторые из них можно игнорировать. Весь необходимый код и вспомогательные файлы лабораторной работы находятся в архиве **МиСИИ_лаб2_3_2024.zip:**

Файлы для редактирования:	
search.py	Здесь будут размещаться все алгоритмы поиска, которые Вы запрограммируете.
searchAgents.py	Здесь будут находиться все Ваши поисковые агенты.
Файлы, которые необходимо просмотреть	
pacman.py	Основной файл, из которого запускают Pacman. Этот файл описывает тип Pacman GameState, который используется в лабораторных работах.
game.py	Логика, лежащая в основе мира Pacman. Этот файл описывает несколько поддерживающих типов, таких как AgentState, Agent, Direction и Grid.
util.py	Полезные структуры данных для реализации алгоритмов поиска.
Поддерживающие файлы, которые можно игнорировать:	
graphicsDisplay.py	Графика Pacman
graphicsUtils.py	Графические утилиты
textDisplay.py	ASCII графика Pacman
ghostAgents.py	Агенты, управляющие привидениями
keyboardAgents.py	Интерфейс клавиатуры для управления игрой
layout.py	Код для чтения файлов схем и хранения их содержимого
autograder.py	Автооцениватель
testParser.py	Парсер тестов автооценивателя и файлы решений
testClasses.py	Общие классы автооценивателя
test_cases/	Папка, содержащая тесты для каждого из заданий (вопросов)
searchTestClasses.py	Специальные тестовые классы автооценивателя для данной лабораторной работы

Во время выполнения данной лабораторной работы необходимо будет дописать код файлов **search.py** и **searchAgents.py**. Пожалуйста, не изменяйте другие файлы в дистрибутиве лабораторной работы.

После загрузки кода (**МиСИИ_лаб2_3_2024.zip**), его распаковки и перехода в соответствующий каталог вы сможете играть в Pacman, набрав в командной строке Python следующее (если используется среда IPython, то набор команды начинается со знака “**!**”):

```
python pacman.py
```

Pacman живет в мире разветвленных лабиринтов и пищевых гранул. Эффективная навигация в этом мире будет первым шагом Pacman в освоении свой области обитания. Самый простой агент, находящийся в **searchAgents.py** и называемый **GoWestAgent**, всегда идет на запад (тривиальный рефлекторный агент). Этот агент может иногда выигрывать:

```
python pacman.py --layout testMaze --pacman GoWestAgent
================
Pacman emerges victorious! Score: 503
Average Score: 503.0
Scores:       503.0
Win Rate:     1/1 (1.00)
Record:       Win
```

Здесь параметр **--layout**, определяют сложность лабиринта, а параметр **-- pacman** – тип используемого агента. Для агента **GoWestAgent** всё становится плохо, когда требуется поворот: он упирается в стену и останавливается. Например,

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

Если Pacman застрял, вы можете выйти из игры, нажав **CTRL+c** на своем терминале. Необходимо будет научить агента преодолевать не только лабиринт **tinyMaze**, но и любой лабиринт по вашему желанию.

Обратите внимание, что **pacman.py** поддерживает ряд параметров, каждый из которых может быть задан длинным (например, **--layout**) или коротким (например, **-l**) способом. Можно просмотреть список всех параметров вызова Pacman и их значения по умолчанию, выполнив команду:

```
python pacman.py –h
```

Все команды вызова Pacman, которые используются в этой лабораторной работе и рассматриваются далее в заданиях, также содержатся в файле **commands.txt** для облегчения копирования и вставки.

2.3. Задания для выполнения

Задание 1. Поиск в глубину

Воспользуемся поисковым агентом **SearchAgent**, реализованным в **searchAgents.py**, который планирует путь в лабиринте, а затем выполняет его пошагово. Ваша задача состоит в том, чтобы реализовать алгоритмы поиска пути для этого агента.

Сначала проверьте правильность работы **SearchAgent,** выполнив команду:

python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

Приведенная выше команда указывает **SearchAgent** использовать **tinyMazeSearch** в качестве алгоритма поиска, который реализован в **search.py.** Pacman должен успешно перемещаться по лабиринту.

Вам необходимо написать полноценные универсальные функции поиска, которые помогут Pacman планировать маршруты. Помните, что поисковый узел должен содержать не только состояние, но и информацию, необходимую для восстановления пути (плана), который приводит в это состояние.

Указание 1: функции поиска должны возвращать список действий, которые приведут агента из стартового состояния в целевое состояние. Все эти действия должны быть разрешенными (должны использоваться допустимые направления, запрет на перемещение через стены).

Указание 2: обязательно используйте структуры данных **Stack, Queue** и **PriorityQueue**, предоставленные в **util.py**. У реализаций этих структур данных есть определенные свойства, которые требуются для совместимости с автооценивателем.

Подсказка: все алгоритмы поиска очень похожи. Алгоритмы для DFS, BFS, UCS отличаются только деталями управления списком **OPEN**. Сконцентрируйтесь на правильном написании кода алгоритма DFS, а остальное должно быть относительно простым. Действительно вместо DFS, BFS, UCS можно реализовать только один обобщенный метод поиска, который настраивается стратегией организации порядка обработки списка **OPEN**, зависящей от алгоритма. (Ваша реализация не обязательно должна быть в этой обобщенной форме).

Реализуйте алгоритм поиска в глубину (DFS) в функции **depthFirstSearch** в файле **search.py.** Напишите версию DFS для поиска пути на графе, которая избегает раскрытия любых уже посещенных состояний. Псевдокод функции **depthFirstSearch** аналогичен псевдокоду функции **breadthFirstSearch**, который был рассмотрен в разделе 2.2.6. При этом список **OPEN** представляет собой не очередь, а стек. Для создания пустого стека используйте вызов **OPEN =util.Stack().**

Написанный код **depthFirstSearch** должен быстро найти решение для следующих лабиринтов:

python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent

Схема лабиринта Pacman (рисунок 2.2) показывает наложение обследованных состояний и порядок, в котором эти состояния обходятся агентом (более яркий красный означает более ранний обход). Ответьте на два вопроса: Вы ожидали такой порядок обхода? На самом ли деле Pacman проходит все обследованные позиции на пути к цели?

Подсказка: если вы используете стек в качестве структуры данных, то решение, найденное вашим алгоритмом DFS для **mediumMaze**, должно иметь длину 130

(при условии, что вставка преемников в **OPEN** выполняется в порядке, формируемом методом **getSuccessors**; вы можете получить длину 246, если будете использовать обратный порядок). Ответьте на два вопроса: Это решение наименьшей стоимости? Если нет, подумайте, что не так с поиском в глубину.

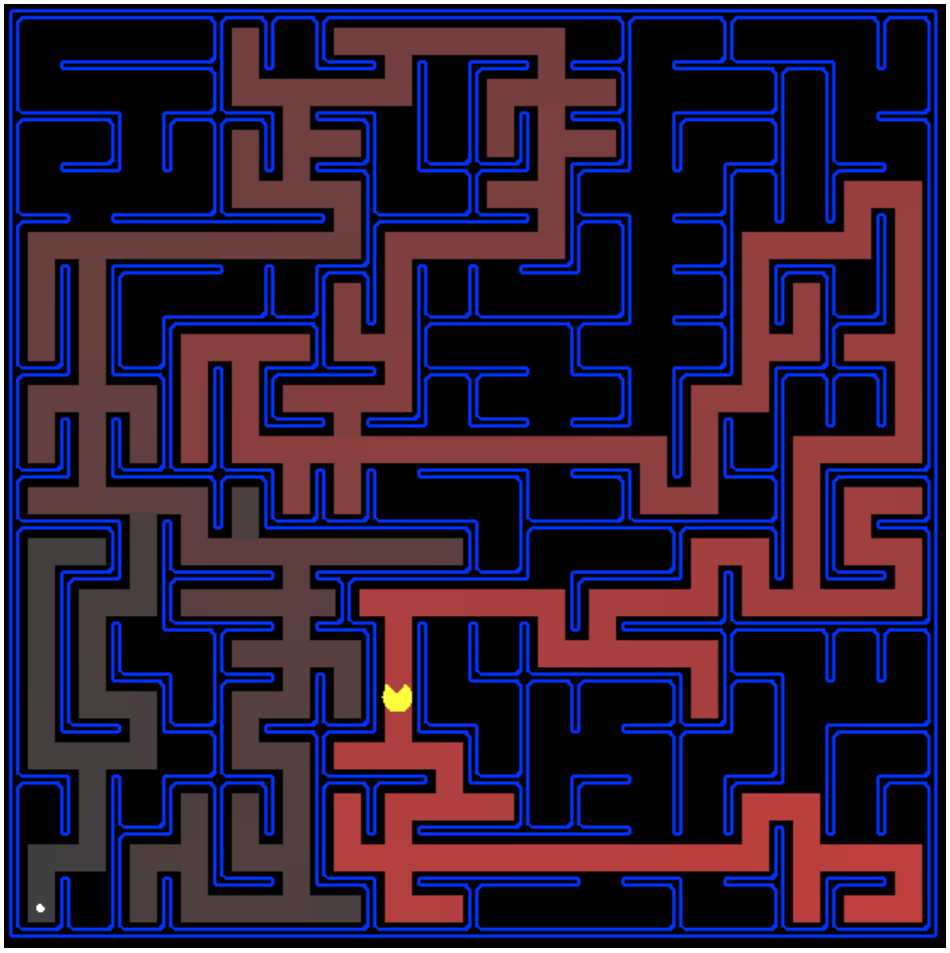


Рисунок 2.2 – Схема большого лабиринта (bigMaze)

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тестовые примеры автооценивателя:

python autograder.py -q q1

Если возникают ошибки, то исправьте их. После успешного прохождения тестов автооценивателя внесите результаты в отчет

.

Задание 2. Поиск в ширину

Реализуйте алгоритм поиска в ширину (BFS) в функции **breadthFirstSearch** в файле **search.py.** Напишите алгоритм поиска пути на графе, который избегает повторного раскрытия вершин. Проверьте свой код так же, как и для поиска в глубину:

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

Ответьте на вопрос: найдет ли BFS решение с наименьшими затратами? Если нет, проверьте свою реализацию.

Подсказка: если Pacman движется слишком медленно, то попробуйте опцию **--frameTime 0**.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тестовые примеры автооценивателя:

python autograder.py -q q2

Если возникают ошибки, то исправьте их. После успешного прохождения тестов автооценивателя внесите результаты в отчет

Задание 3. Поиск на основе алгоритма равных цен

Хотя BFS найдет путь к цели с наименьшим количеством действий, можно попытаться найти путь, который является «лучшим» с точкт зрения иных критериев. Рассмотрим лабиринты **mediumDottedMaze** и **mediumScaryMaze**.

Изменяя функцию стоимости, мы можем побудить Pacman находить разные пути. Например, мы можем назначать большую цену за опасные шаги в областях, рядом с призраками, или меньшую цену за шаги в областях, богатых едой, и рациональный агент Pacman должен корректировать свое поведение.

Реализуйте алгоритм равных цен для поиска пути на графе в функции **uniformCostSearch** в файле **search.py**. Рекомендуется просмотреть файл **util.py** для ознакомления с некоторыми структурами данных, которые могут быть полезны.

Теперь вы должны наблюдать успешное поведение агента во всех трех лабиринтах, где все агенты являются агентами, функционирующими на основе алгоритма UCS, которые отличаются только используемой функцией стоимости (агенты и функции стоимости уже написаны):

python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

Примечание. Вы должны получить очень низкие и очень высокие стоимости путей для агентов **StayEastSearchAgent** и **StayWestSearchAgent**, соответственно, из-за их функций экспоненциальной стоимости (подробности см. в **searchAgents.py**).

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тестовые примеры автооценивателя:

python autograder.py -q q3

Если возникают ошибки, то исправьте их. После успешного прохождения тестов автооценивателя внесите результаты в отчет

2.4. Порядок выполнения лабораторной работы

2.4.1. Изучить по лекционному материалу и учебным пособиям [1-3] методы слепого поиска решений задач в пространстве состояний. Если Вы еще не распаковали архив **МиСИИ_лаб2_3_2024.zip** с файлами лабораторной работы, то разаривируйте его в отдельную папку.

2.4.2. Изучить структуры данных **Stack, Queue** и **PriorityQueue**, предоставленные в модуле **util.py**.

2.4.3. Изучить методы среды AI Pacman: **problem.getStartState()** , **problem.isGoalState(), problem.getSuccessors().** Для этого проверить выполнение команды

```
python pacman.py -l tinyMaze -p SearchAgent
```

для случая, когда реализация функции **depthFirstSearch** содержит только вызовы операторов печати

```
print("Start:", problem.getStartState())
print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
print("Start's successors:", problem.getSuccessors(problem.getStartState()))
```

Разобраться с типами значений, возвращаемых методами и использовать их при написании кода, реализующего алгоритмы поиска DFS, BFS, UCS. Обратите внимание, что состояние-приемник, возвращаемое методом **getSuccessors()**, представляет собой кортеж **((x,y), действие, стоимость),** содержащий: **(x,y)** – координаты узла-приемника; **действие** – команда (‘East’, 'West’, 'North’, 'South’), обеспечивающая переход в состояние приемник; **стоимость** - стоимость перехода в состояние-приемник.

2.4.4. Определить в соответствии с заданиями 1-3 раздела 2.3 функции, реализующие алгоритмы поиска DFS, BFS, UCS. При реализации алгоритмов поиска рекомендуется использовать псевдокод из раздела 2.2.6. Для реализации списка **OPEN** в алгоритме UCS следует использовать очередь с приоритетами **PriorityQueue**. Рекомендуется для редактирования функций и выполнения кода использовать интегрированную среду (IDE). Следует выполнять вставку кода, определяемых функций, после строки:

```
"*** ВСТАВЬТЕ ВАШ КОД СЮДА***"
```

Никакие другие функции архива **МиСИИ_лаб2_3_2024.zip** не менять.

2.4.5. Зафиксировать результаты использования функций для всех лабиринтов, указанных в заданиях 1-3. Ответить письменно на предлагаемые в заданиях 1-3 вопросы.

2.4.6. Выполнить с помощью **autograder.py** автооценивание заданий 1-3. При обнаружении ошибок отредактировать код. Результаты автооценивания внести в отчет.

2.4.7. Оценить эффективность используемых методов поиска по критериям временной и пространственной сложности.

2.5. Содержание отчета

Цель работы, краткий обзор методов слепого поиска решений задач в пространстве состояний, описание представления задачи в AI Pacman (описание состояний, операторов, начального и конечного состояний, критериев достижения цели), представление пространства состояний в виде графа, тексты реализованных функций с комментариями, полученные результаты для разных лабиринтов и их анализ, результаты автооценивания, выводы по проведенным экспериментам с разными алгоритмами слепого поиска и разными лабиринтами.

2.6. Контрольные вопросы

2.6.1. Назовите основные способы представления задач в ИИ?

2.6.2. Определите состояния, операторы преобразования состояний, функции стоимости для следующих задач:

а) задача о коммивояжере;

б) задача о миссионерах и каннибалах;

в) задача о раскраске карт.

2.6.3. Сформулируйте для задачи поиска пути и задачи поедания всех пищевых гранул в AI Pacman, что представляют собой состояния, действия, функция-приемник, функция проверки цели.

2.6.4. Для задачи о двух кувшинах емкостью 5 литров и 2 литра, построить дерево поиска, если требуется налить во второй кувшин ровно 1 литр воды.

2.6.5. Напишите на псевдоязыке процедуры поиска в ширину и глубину, объясните их различие с алгоритмической точки зрения.

2.6.6. Напишите на псевдоязыке процедуру поиска в соответствии с алгоритмом равных цен.

2.6.7. Объясните назначение функций **problem.getStartState()**, **problem.isGoalState(), problem.getSuccessors()** среды AI Pacman и приведите примеры значений, возвращаемых этими функциями.

2.6.8. Напишите на языке Python функцию, реализующую поиск в глубину применительно к среде AI Pacman.

2.6.9. Сформулируйте принципы поиска, используемые в алгоритмах поиска в глубину: с возвратом, с ограничением глубины и с итеративным углублением.

2.6.10. Определите критерии полноты, оптимальности, минимальности, пространственной и временной сложности.

2.6.11. Сравните основные алгоритмы слепого поиска в пространстве состояний по критериям полноты, оптимальности, пространственной и временной сложности.

3. ЛАБОРАТОРНАЯ РАБОТА № 3 «ИССЛЕДОВАНИЕ ИНФОРМИРОВАННЫХ МЕТОДОВ ПОИСКА РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»

3.1. Цель работы

Исследование информированных методов поиска решений задач в пространстве состояний, приобретение навыков программирования интеллектуальных агентов, планирующих действия на основе методов эвристического поиска решений задач.

3.2. Краткие теоретические сведения

3.2.1 Общая характеристика методов информированного поиска

Основная идея таких методов состоит в использовании дополнительной информации для ускорения процесса поиска. Эта дополнительная информация формируется на основе эмпирического опыта, догадок и интуиции исследователя, т.е. **эвристик.** Использование эвристик позволяет сократить количество просматриваемых вариантов при поиске решения задачи, что ведет к более быстрому достижению цели.

В алгоритмах эвристического поиска список открытых вершин упорядочивается по возрастанию некоторой ***оценочной функцией***, формируемой на основе эвристических правил. Оценочная функция может включать две составляющие, одна из которых называется эвристической и характеризует близость текущей и целевой вершин. Чем меньше значение эвристической составляющей оценочной функции, тем “ближе” рассматриваемая вершина к целевой вершине. В зависимости от способа формирования оценочной функции выделяют следующие алгоритмы эвристического поиска: алгоритм “подъема на гору”, алгоритм глобального учета соответствия цели, А-алгоритм [1-3]. Наиболее общим является А-алгоритм.

3.2.2. А - алгоритм

А-алгоритм похож на алгоритм равных цен, но в отличие от него учитывает при раскрытии вершин, как уже сделанные затраты, так и предстоящие затраты. В этом случае оценочная функция имеет вид:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

где $\hat{g}(n)$ – оценка стоимости пути из начальной вершины в вершину n, которая вычисляется в соответствии с (2.2); $\hat{h}(n)$ – **эвристическая** оценка стоимости кратчайшего пути из вершины n в целевую вершину (предстоящие затраты). Чем меньше значение $\hat{h}(n)$, тем перспективнее путь, на котором находится вершина n. В ходе поиска раскрываются вершины с минимальным значением оценочной функции $\hat{f}(n)$.

Готовых рецептов в отношении построения эвристической составляющей оценочной функции не существует. При решении каждой конкретной задачи используется ранее накопленный опыт решения подобных задач. Например, для **игры в восемь** это может быть количество фишек, которые находятся не на своем месте или сумма расстояний каждой фишки от текущего до целевого расположения.

Схема А*- алгоритма соответствует алгоритму равных цен. При этом оценки $\hat{f}(n)$ могут менять свои значения в процессе поиска. Это приводит к тому, что вершины из списка **CLOSED** могут перемещаться обратно в список **OPEN**. Ниже приведен псевдокод функции, выполняющей поиск на графе в соответствии с А-алгоритмом [1]:

def aStarSearch **(problem):**
Определить стартовую вершину**: start = problem.getStartState()**
Поместить стартовую вершину в список OPEN**: OPEN.push(start)**
CLOSED = [ ]
Путь = [ ]
while not OPEN.isEmpty() :
node = OPEN.pop()
If node == 'целевая вершина'**: return Путь**
CLOSED = CLOSED.append(node)
Раскрыть **node** и для всех дочерних вершин n_i вычислить оценку $\hat{f}(n, n_i) = \hat{g}(n, n_i) + \hat{h}(n_i)$
Поместить все дочерние вершины, отсутствующие в списке **CLOSED** или **OPEN,** в список **OPEN,** связав с каждой дочерней вершиной указатель на **node**
Для дочерних вершин n_i**,** которые уже содержатся в **OPEN, с**равнить оценки $\hat{f}(n,n_i)$ и $\hat{f}(n_i)$**,** если $\hat{f}(n,n_i) < \hat{f}(n_i)$**,** то связать с вершиной n_i новую оценку $\hat{f}(n,n_i)$ и указатель на вершину **node**
Если вершина n_i содержится в списке **CLOSED** и $\hat{f}(n,n_i) < \hat{f}(n_i)$**,** то связать с вершиной n_i новую оценку $\hat{f}(n,n_i)$**,** переместить её в список **OPEN** и установить указатель на **node;**
Упорядочить список **OPEN** по возрастанию $\hat{f}(n_i)$**;**
return 'НЕУДАЧА'

3.2.3. Свойства А-алгоритма

Свойства А-алгоритма существенно зависят от условий, которым удовлетворяет или не удовлетворяет эвристическая часть оценочной функции $\hat{f}(n)$ [1]:

1) А-алгоритм соответствует алгоритму равных цен, если *h(n)*=0;

2) А-алгоритм **гарантирует оптимальное решение**, если $\hat{h}(n) \le h(n)$; в этом случае он называется **А* – алгоритмом**. А*-алгоритм недооценивает затраты на пути из промежуточной вершины в целевую вершину или оценивает их правильно, но никогда не переоценивает;
3) А-алгоритм обеспечивает однократное раскрытие вершин, если выполняется ***условие монотонности*** $\hat{h}(n_i) \le \hat{h}(n_j) + c(n_i, n_j)$, где n_i – родительская вершина; n_j – дочерняя вершина; $c(n_i, n_j)$ – стоимость пути между вершинами n_i и n_j;
4) алгоритм A_1* является эвристически более сильным, чем алгоритм A_2* при условии $\hat{h}_1(n) > \hat{h}_2(n)$. Эвристически более сильный алгоритм A_1* в большей степени сокращает пространство поиска;
5) А*-алгоритм полностью информирован, если $\hat{h}(n) = h(n)$. В этом случае никакого поиска не происходит и приближение к цели идет по оптимальному пути;
6) при $\hat{h}(n) > h(n)$ А-алгоритм не гарантирует получение оптимального решения, однако часто решение получается быстро.

Эффективность поиска с помощью **А***-алгоритма может снижаться из-за того, что вершина, находящаяся в списке **CLOSED**, может попадать обратно в список **OPEN** и затем повторно раскрываться.

Чтобы **А***-алгоритм не раскрывал несколько раз одну и ту же вершину необходимо, чтобы выполнялось ***условие монотонности***

$$\hat{h}(n_i) \le \hat{h}(n_j) + c(n_i, n_j),$$

где n_i – родительская вершина; n_j – дочерняя вершина; $c(n_i, n_j)$ – стоимость пути между вершинами n_i и n_j. Монотонность предполагает, что не только не переоценивается стоимость $\hat{h}(n)$ оставшегося пути из n до цели, но и не переоцениваются стоимости ребер между двумя соседними вершинами.

Условие монотонности поглощает условие гарантированности (допустимости). *Если условие монотонности соблюдается для всех дочерних вершин, то можно доказать, что в тот момент, когда раскрывается некоторая вершина n,* ***оптимальный путь*** *к ней уже найден.* Следовательно, оценочная функция для данной вершины в дальнейшем не меняет своих значений, и никакие вершины из списка **CLOSED** в список **OPEN** не возвращаются.

Если соблюдается условие монотонности, то значения оценочной функции $f(n)$ вдоль любого пути являются **неубывающими**. Тот факт, что стоимости вдоль любого пути являются не убывающими, означает что в пространстве состояний могут быть очерчены **контуры равных стоимостей**. Поэтому **А***-алгоритм проверяет все узлы в контуре меньшей стоимости, прежде чем перейдет к проверке узлов следующего контура.

А*-алгоритм (при выполнении условия монотонности) является **полным, оптимальным и оптимально эффективным** (не гарантируется развертывание меньшего кол-ва узлов с помощью любого иного оптимального алгоритма).

Временная сложность A*-алгоритма по-прежнему остается **экспоненциальной**, т.е количество раскрываемых узлов в пределах целевого контура пространства состояний все еще зависит экспоненциально от глубины решения. По этой причине на практике стремление находить оптимальное решение не всегда оправдано. Иногда вместо этого целесообразно использовать варианты **A-**поиска, позволяющие быстро находить квазиоптимальные решения, или разрабатывать более точные эвристические функции, но не строго допустимые.

Так как при **A***-поиске хранятся все раскрытые узлы, фактические **ресурсы пространства** исчерпаются гораздо раньше, чем временные ресурсы. С этой целью применяют ***A*-алгоритм с итеративным углублением (IDA*).*** Применяемым условием остановки здесь служит **f**-стоимость, а не глубина. Преодолеть проблему пространственной сложности за счет небольшого увеличения времени выполнения позволяют также алгоритмы **RBFS** и **MA*** [4].

3.3. Задания для выполнения

Задание 1. A*- поиск

Реализуйте A*-алгоритм на графе состояний, используя шаблон функции **aStarSearch** в файле **search.py**. Функция **aStarSearch** принимает в качестве аргумента эвристическую функцию. Эвристическая функция имеет два аргумента: состояние агента (основной аргумент) и задача (problem) (для справочной информации). Функция **nullHeuristic** в **search.py** является тривиальным примером эвристической функции.

Протестируйте свою реализацию A*-поиска на задаче поиска пути в лабиринте, используя эвристику манхэттенского расстояния (уже реализованную как **manhattanHeuristic** в **searchAgents.py**):

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattan-
Heuristic
```

Вы должны увидеть, что A*- алгоритм находит оптимальное решение немного быстрее, чем поиск в соответствии с алгоритмом равных цен (он раскрывает около 549 узлов по сравнению с 620 узлами, из-за учета приоритета узлов числа могут немного отличаться).

Проверьте результаты поиска на лабиринте **openMaze**. Что можно сказать о различных стратегиях поиска?

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация A*-поиска все тестовые примеры автооценивателя:

```
python autograder.py -q q4
```

Задание 2. Поиск всех углов

Настоящая сила A*-поиска станет очевидной только при решении более сложной задачи. Сформулируем новую проблему и разработаем эвристику для ее решения.

В углах лабиринта есть четыре точки, по одной в каждом углу. Необходимо найти кратчайший путь, который связан с посещением всех четырех углов (независимо от того, есть ли в лабиринте еда или нет). Обратите внимание, что для некоторых лабиринтов, таких как **tinyCorners**, кратчайший путь не всегда ведет к ближайшей грануле в первую очередь!

Рисунок 2.3. – Задача поиска углов

Подсказка: кратчайший путь через **tinyCorners** состоит из 28 шагов.
Примечание. Обязательно выполните задание 2 предыдущей лабораторной работы, прежде чем решать это задание

Реализуйте задачу поиска углов, дописав участки кода в определении класса **CornersProblem** в файле **searchAgents.py**. Вам нужно будет выбрать такое представление состояния, которое кодирует всю информацию, необходимую для определения достижения цели: посетил ли агент все четыре угла.

Протестируйте поискового агента, выполнив команды:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

Чтобы получить высокую оценку за выполнение задания, вам необходимо определить абстрактное представление состояния, которое не содержит несущественную информацию (например, положение призраков, где находится дополнительная еда и т. п.). В частности, не используйте Pacman **GameState** в качестве состояния поиска. Ваш код будет очень медленным.

Подсказка 1. Единственные части игрового состояния, на которые вам нужно ссылаться в своей реализации, - это начальная позиция Pacman и расположение четырех углов.

Подсказка 2: при написании кода **getSuccessors** не забудьте добавить потомков в список преемников со стоимостью 1.

Наша реализация **breadthFirstSearch** расширяет почти 2000 поисковых узлов на задаче **mediumCorners.** Однако эвристика (используемая в A*-поиске) может уменьшить этот объем.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация агента, выполняющего поиск углов, все тесты автооценивателя:

python autograder.py -q q5

Задание 3. Эвристика для задачи поиска углов

Реализуйте нетривиальную монотонную эвристику для задачи поиска углов в методе **cornersHeuristic** класса **CornersProblem.** Проверьте реализацию, выполнив команду:

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

Здесь **AStarCornersAgent** - это сокращение (ярлык) для

-p SearchAgent -a fn = aStarSearch, prob = CornersProblem, heuristic = cornersHeuristic

Гарантированность (*admissibility)* **против монотонности (***consistency***)**: помните, эвристики - это просто функции, которые принимают состояния поиска и возвращают числа, которые дают прогноз стоимости достижения ближайшей цели. Более эффективная эвристика вернет значения, близкие к фактическим затратам. Чтобы быть гарантирующими (допустимыми), эвристические значения должны быть нижними границами фактических затрат кратчайшего пути к ближайшей цели (и неотрицательными). Чтобы быть монотонными (согласованными), они должны дополнительно обеспечивать выполнение условия: если действие имеет стоимость *c*, то выполнение этого действия может вызвать только снижение значения эвристики не более чем на *c*.

Помните, что гарантированности (допустимости) недостаточно, чтобы обеспечить правильность поиска по графу - вам нужно более строгое условие монотонности. Однако допустимые эвристики часто также монотонны (согласованны). Поэтому обычно проще всего начать с мозгового штурма допустимых эвристик. Если у вас есть допустимая эвристика, которая хорошо работает, вы также можете проверить, действительно ли она согласована. Единственный способ проверить согласованность - это предъявить доказательство. Однако несогласованность часто можно обнаружить, убедившись, что для каждого расширяемого узла его последующие узлы имеют равные или большие значения *f*. Кроме того, если алгоритмы UCS и A* когда-либо возвращают пути разной длины, ваша эвристика не согласована.

Тривиальные эвристики - это те, которые возвращают ноль везде (UCS), и эвристики, которые вычисляют истинную стоимость. Первая не снизит временную сложность, а вторая приведет к таймауту автооценивателя. Вам нужна нетривиальная эвристика, которая сокращает общее время вычислений, хотя для этого задания

автооцениватель будет проверять только количество узлов (помимо обеспечения разумного ограничения по времени).

Оценивание: ваша эвристика должна быть нетривиальной, неотрицательной и согласованной, чтобы получить какие-либо баллы за выполнение задания. Убедитесь, что ваша эвристика возвращает 0 при каждом целевом состоянии и никогда не возвращает отрицательное значение. В зависимости от того, сколько узлов раскроет ваша эвристика, вы получите следующие оценки за выполнение задания:

Число раскрываемых узлов	Оценка
более 2000	0/3
максимум 2000	1/3
максимум 1600	2/3
максимум 1200	3/3

Помните: если ваша эвристика не будет монотонной, вы не получите баллов!

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация эвристической функции все тесты автооценивателя

python autograder.py -q q6

Задание 4. Задача поедания всех гранул

Теперь мы решим сложную задачу поиска: агент должен съесть всю еду за минимальное количество шагов. Для этого нам понадобится новое определение задачи поиска, которое формализует поедание всех пищевых гранул. Эта задача реализуется классом **FoodSearchProblem** в **searchAgents.py**.

Решение определяется как путь, вдоль которого собирается вся еда в мире Pacman. Для данного задания не учитываются призраки или энергетические гранулы; решения зависят только от расположения стен, пищевых гранул и агента. (Конечно, призраки могут ухудшить решения! Мы вернемся к этому в следующих лабораторных работах.) Если будут правильно написаны общие методы поиска, то A*-поиск с нулевой эвристикой (эквивалент поиска с равномерной стоимостью) должен быстро найти оптимальное решение для **testSearch** без изменения кода с вашей стороны (общая стоимость пути 7). Проверьте:

python pacman.py -l testSearch -p AStarFoodSearchAgent

Здесь **AStarFoodSearchAgent** - это сокращение (ярлык) для

-p SearchAgent -a fn = astar, prob = FoodSearchProblem, эвристика = foodHeuristic

Вы должны обнаружить, что алгоритм UCS начинает замедляться даже в случае простого лабиринта **tinySearch**.

Примечание. Обязательно выполните задание 1, прежде чем работать над заданием 4.

Допишите код в функции **foodHeuristic** в файле **searchAgents.py**, определив монотонную (согласованную) эвристику для класса **FoodSearchProblem**. Проверьте работу агента на сложной задаче поиска (требует времени):

python pacman.py -l trickySearch -p AStarFoodSearchAgent

Любая нетривиальная неотрицательная согласованная эвристика, разработанная Вами получит 1 балл. Убедитесь, что ваша эвристика возвращает 0 при каждом целевом состоянии и никогда не возвращает отрицательное значение. В зависимости от того, сколько узлов будет раскрывать ваша эвристика, вы получите дополнительные баллы:

Число раскрываемых узлов	Оценка
более 15000	1/4
максимум 15000	2/4
максимум 12000	3/4
максимум 9000	4/4
максимум 7000	5/4 (трудно)

Помните: если ваша эвристика немонотонна, вы не получите баллов. Сможете ли вы решить **mediumSearch** за короткое время? Если да, то это либо впечатляющий результат, либо ваша эвристика немонотонна.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тесты автоценивателя:

python autograder.py -q q7

Задание 5. Субоптимальный поиск

Иногда даже с помощью A*-поиска при хорошей эвристике найти оптимальный путь через все точки накладно. В таких случаях можно просто выполнить быстрый поиск “достаточно” хорошего пути. В этом задании необходимо реализовать агента, который всегда жадно ест ближайшую гранулу. Такой агент **ClosestDotSearchAgent** реализован в файле **searchAgents.py**, но в нем отсутствует ключевая функция, которая находит путь к ближайшей точке.

Реализуйте функцию **findPathToClosestDot** в **searchAgents.py**. Проверьте решение:

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

Агент выполнит поиск субоптимального пути в этом лабиринте со стоимостью пути 350.

Подсказка: самый быстрый способ завершить **findPathToClosestDot** - это заполнить в классе **AnyFoodSearchProblem** функцию проверки достижения цели **isGoalState**. А затем завершить определение **findPathToClosestDot** с помощью соответствующей функции поиска, написанной ранее. Решение должно получиться очень коротким!

Ваш агент **ClosestDotSearchAgent** не всегда будет находить кратчайший путь через лабиринт. Убедитесь, что вы понимаете, почему, и попробуйте придумать небольшой пример, где многократный переход к ближайшей точке не приводит к нахождению кратчайшего пути для съедания всех точек.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тесты автооценивателя:

```
python autograder.py -q q8
```

3.4. Порядок выполнения лабораторной работы

3.4.1. Изучить по лекционному материалу или учебным пособиям [1-3] методы информированного поиска решений задач в пространстве состояний.

3.4.2. Использовать для выполнения лабораторной работы файлы из архива **МиСИИ_лаб2_3_2024.zip.** Изучить структуру данных, соответствующую очереди с приоритетами **PriorityQueue**, реализованную в модуле **util.py**.

3.4.3. Изучите эвристическую функцию, вычисляющую манхэттенское расстояние:

```
def manhattanHeuristic(position, problem, info={}):
    "The Manhattan distance heuristic for a PositionSearchProblem"
    xy1 = position                #текущая позиция
    xy2 = problem.goal            #целевая позиция
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])
```

Функция вычисляет сумму абсолютных разностей координат текущей позиции и целевой позиции для задачи **PositionSearchProblem**.

3.4.4. Для реализации A*- алгоритма в соответствии с заданием 1 используйте псевдокод из раздела 3.2.2. При этом создайте список открытых вершин в виде экземпляра очереди с приоритетом элементов: **OPEN = util.PriorityQueue().** Для реализации части псевдокода, связанной со вставкой состояний-приемников в список **OPEN** используйте метод **OPEN.update,** определенный в классе **PriorityQueue().** Метод **update** выполняет вставку или обновление элементов в очереди с учетом их приоритетов. В качестве приоритета используйте значение оценочной функции *f*. В этом случае не нужно будет выполнять упорядочение **OPEN** по возрастанию оценочной функции. Поиск состояний-приемников для узла **node** реализуется с использованием метода **problem.getSuccessors(node)**.

3.4.5 Для реализации задания 2 необходимо дописать код следующих методов в классе **CornersProblem** в файле **searchAgents.py: __init__, getStartState, isGoalState, getSuccessors.** При этом необходимо выбрать подходящее представление состояний для задачи поиска углов. Удобно представить состояние **state** в виде кортежа ((x,y),((x1,y1),(x2,y2),…)), где x,y – координаты текущей позиции агента, x1,y1 и т.д – координаты посещенных углов.

Рекомендуется добавить в конструктор класса **CornersProblem** присваивание значения атрибуту **self.startingGameState= startingGameState**.

Добавьте в метод **getStartState** возврат начального состояния **self.startingPosition** и пустого кортежа **()**. Пустой кортеж в дальнейшем будет заполняться позициями-кортежами посещенных углов.

Метод **isGoalState** для выбранного представления состояния задачи должен просто проверять длину второго кортежа представления. Если она будет равна 4, то текущее состояние – целевое состояние.

При реализации метода **getSuccessors** необходимо внимательно прочитать следующий комментарий внутри цикла **for** по возможным действиям (направлениям перемещения) агента:

```
# Добавьте состояние-приемник в список приемников, если действие является
# допустимым
# Ниже фрагмент кода, который выясняет, не попадает ли новая позиция на
# стену лабиринта:
    #   x,y = currentPosition
    #   dx, dy = Actions.directionToVector(action)
    #   nextx, nexty = int(x + dx), int(y + dy)
    #   hitsWall = self.walls[nextx][nexty]
```

Здесь **currentPosition = state[0].**

Необходимо воспользоваться этой частью кода. Если координаты новой позиции не попадают на стену (верно **not hitsWall)**, то следует сформировать новое состояние в соответствии с выбранным представлением (см. выше):

```
new_state = ((nextx, nexty), state[1])
```

А если координаты новой позиции **nextx, nexty** сооответствуют углу (т.е. содержатся в **self.corners**) и этот угол еще не посещался (т.е. отсутствуют в **state[1]**), то новое состояние должно получить следующее значение:

```
new_state = ((nextx, nexty),  (state[1] + ((nextx, nexty), ))
```

В итоге если действие не приводит к столкновению со стеной, то новое состояние должно быть добавлено в список-приемников:

```
successors.append((new_state, action, 1))
```

3.4.6. В задании 3 необходимо определить эвристическую функцию **cornersHeuristic** для задачи поиска углов. Возможный вариант эвристической функции: находить непосещенные углы для заданного состояния

```
corners = problem.corners   # координаты углов
position = state[0]         # текущая позиция
touchedcorners = state[1]   # посещенные углы
# список непосещенных углов -- разность 2-х множеств
untouchedcorners = list(set(corners).difference(set(touchedcorners)))
```

и вычислять манхэттенское расстояние **(abs(corner[0] - position[0]) + abs(corner[1] - position[1]))** от позиции агента до ближайшего непосещенного угла, виртуально обходя таким образом ближайшие углы; в качестве значения эвристической функции возвращать сумму виртуальных ближайших расстояний.

Возможны и другие варианты эвристик, например, основанные на вычислении расстояний непосредственно по лабиринту с использованием уже реализованных алгоритмов поиска путей. (см. ниже п.3.4.7)

3.4.7. При решении задания 4 - поедание всех пищевых гранул – требуется определить нетривиальную монотонную эвристику в методе **foodHeuristic(state, problem)** класса **FoodSearchProblem**. Рекомендуется сначала придумать допустимую эвристику. Обычно допустимые эвристики также оказываются монотонными.

Если при использовании A*-алгоритма будет найдено решение, которое хуже, чем поиск в соответствии алгоритмом равных цен, ваша эвристика немонотонная и, скорее всего, недопустима. С другой стороны, немонотонные эвристики могут найти оптимальные решения, поэтому будьте осторожны.

Состояние в рассматриваемой задаче представляется в виде кортежа **(pacmanPosition, foodGrid)**, где **foodGrid** относится к типу **Grid** (см. **game.py**). Чтобы получить список координат точек размещения еды можно вызвать **foodGrid.asList()**.

Если нужен будет доступ к информации о стенах можно сделать запрос в виде **problem.walls**, который вернет объект типа **Grid** с расположением стен. Если необходимо будет сохранять информацию для повторного использования, то можно использовать словарь **problem.heuristicInfo**. Например, если вы хотите посчитать стены только один раз и сохранить это значение, используйте запрос: **problem.heuristicInfo ['wallCount'] = problem.walls.count ()**

Написание кода эвристики начните с простой проверки: если длина списка **foodGrid.asList()** равна нулю, то верните **0**. Если указанный список не пустой, то можно, например, вычислить “расстояния” от текущей позиции до каждой пищевой гранулы. Таким образом можно будет спрогнозировать затраты на достижение целевого состояния – поедание всех гранул. Помните, прогнозные затраты не должны превышать реальных затрат. Один из вариантов вычисления расстояний предоставляет функция **mazeDistance(point1, point2, gameState)** в файле **searchAgents.py**. Эта функция строит путь по лабиринту между точками **point** и возвращает его длину.

3.4.8. В задании 5 необходимо реализовать функцию субоптимального поиска **findPathToClosestDot**, обеспечивающей реализацию агента **ClosestDotSearchAgent,** осуществляющего жадный поиск.

Как указано в задании, сначала определите функцию **isGoalState(state)** класса **AnyFoodSearchProblem**. Нужно просто вернуть результат проверки принадлежности выбранной текущей точки **x,y=state** списку, возвращаемому методом **food.asList()** объекта типа **AnyFoodSearchProblem**.

После этого путь до ближайшей точки в **findPathToClosestDot** может быть найден, например, с помощью вызова алгоритма равных цен для задачи **AnyFoodSearchProblem(gameState).**

3.4.9. Для всех заданий необходимо выполнить проверку тестов автооценивателя, результаты прохождения тестов внести в отчет.

3.5. Содержание отчета

Цель работы, краткий обзор методов информированного поиска решений задач в пространстве состояний, описание свойств A*- алгоритма, тексты реализованных функций с комментариями в соответствии с заданиями 1-5, результаты выполнения поиска для разных задач и алгоритмов и их анализ, результаты автооценивания, выводы по проведенным экспериментам с разными алгоритмами информированного поиска.

3.6. Контрольные вопросы

3.6.1. Что называют эвристикой?

3.6.2. Объясните основной принцип построения процедур эвристического поиска. Запишите вид оценочной функции и объясните её составляющие.

3.6.3. Напишите на псевдоязыке процедуру поиска в соответствии с A-алгоритмом.

3.6.4. Сформулируйте алгоритм подъема в гору.

3.6.5. Сформулируйте алгоритм глобального выбора первой наилучшей вершины.

3.6.6. Почему в A-алгоритме возможен возврат вершин из списка закрытых вершин в список открытых вершин? Приведите примеры.

3.6.7. Сформулируйте и объясните свойства A-алгоритма.

3.6.8. Какой A-алгоритм называют гарантирующим (допустимым)?

3.6.9. Сформулируйте и объясните условие монотонности.

3.6.10. Сформулируйте эвристику манхэттенского расстояния.

3.6.11. Сравните алгоритмы слепого и эвристического поиска по критерию гарантированности получения результата и эффективности поиска.

4. ЛАБОРАТОРНАЯ РАБОТА № 4 «ИССЛЕДОВАНИЕ МЕТОДОВ МУЛЬТИАГЕНТНОГО ПОИСКА»

4.1. Цель работы

Исследование состязательных методов поиска в мультиагентных средах, приобретение навыков программирования интеллектуальных состязательных агентов, возвращающих стратегию поиска на основе оценочных функций.

4.2. Краткие теоретические сведения

4.2.1 Поиск решений в игровых программах

Существует много разных типов игр. В играх могут выполняться детерминированные или стохастические (вероятностные) ходы, в них могут принимать участие один или несколько игроков.

Первый класс игр, который мы рассмотрим, - это **детерминированные игры с нулевой суммой** (шашки, шахматы, го и др.). Такие игры характеризуются полной информацией о текущей игровой ситуации (имеются игры с неполной информацией), где два игрока-противника по очереди делают ходы. Успех одного игрока – такая же по величине потеря для другого игрока. Самый простой способ представить себе такие игры - это их определение с помощью единственной переменной, значение которой агент-игрок пытается максимизировать, а его противник пытается минимизировать. Например, в игре Pacman такая переменная соответствует набранным баллам, которые Pacman пытается максимизировать, поедая гранулы, в то время как призраки пытаются свести к минимуму эти баллы, съедая агента. Фактически игроки в этом случае соревнуются (состязаются), поэтому поиск в игровых программах называют **состязательным поиском**.

Сложность поиска в играх весьма высока, так как вершины дерева игры имеют высокую степень ветвления. Поэтому при поиске по дереву игры необходимо: прогнозировать граничную глубину поиска; оценивать перспективность позиций игры с помощью оценочных функций. Для этого в каждой позиции игры формируется дерево возможных продолжений игры, имеющее определенную глубину, и с помощью некоторой оценочной функции вычисляются оценки концевых вершин такого дерева. Затем полученные оценки распространяются вверх по дереву, и корневая вершина, соответствующая текущей позиции, получает оценку, позволяющую оценить перспективность того или иного хода из этой вершины.

В отличие от рассмотренных ранее методов поиска, которые возвращали исчерпывающий план, состязательный поиск возвращает **стратегию или политику**, которая просто рекомендует наилучший возможный ход с учетом некоторой конфигурации игры.

4.2.2. Минимаксный поиск.

В соответствии с минимаксным методом поиска вместо полного просмотра дерева игры обследуется лишь его небольшая часть. В этом случае говорят, что дерево игры подвергается **подрезке.** Простейший способ подрезки — это просмотр дерева игры на определенную глубину. Поэтому **дерево поиска** — это только верхняя часть дерева игры.

Различают два вида оценок вершин дерева поиска: статические и динамические [1]. **Статические оценки** приписываются терминальным вершинам (состояниям) дерева игры. Вычисление этих оценок реализуются с помощью **функции полезности** (utility function). **Динамические оценки** получаются при распространении статических оценок вверх по дереву. Метод, которым это достигается, называется **минимаксным**. Для вершины, в которой ход выполняет игрок (МАКС), выбирается наибольшая из оценок вершин нижнего уровня (т.е. уровня МИН'а). Для вершины, в которой ход выполняет противник, выбирается наименьшая из оценок дочерних вершин. На рисунке 4.1 изображен пример распространения оценок вверх по дереву в соответствии с указанным минимаксным принципом. Из рисунка 4.1 следует, что лучшим ходом МАКС'а в вершине *S* будет ход *S-D*, а лучшим ходом МИН'а в вершине *D* будет ход *D-L*.

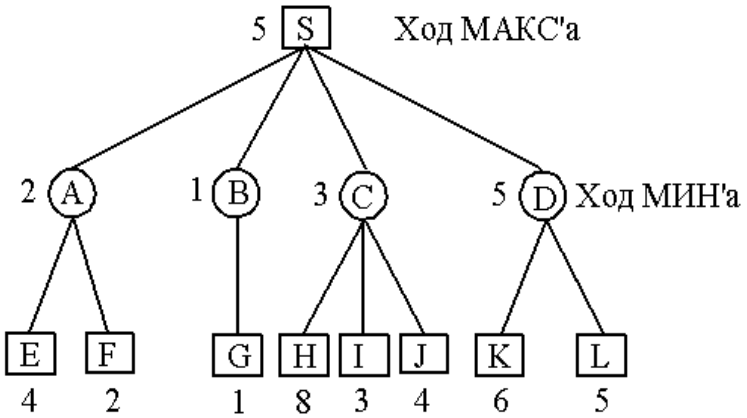


Рисунок 4.1 – Распространение оценок при минимаксной игре

Если обозначить статическую оценку в вершине (состоянии) *s* через *utility(s)*, а динамическую – через *V(s)*, то формально оценки, приписываемые вершинам (состояниям) в соответствии с минимаксным принципом, можно записать так [1]:

$$V(s)=utility(s),$$

если *s* – терминальная вершина (состояние) дерева поиска;

$$V(s)=\max_i V(s_i),$$

если *s* – вершина (состояние) с ходом МАКС'а;

$$V(s)=\min_i V(s_i),$$

если s – вершина (состояние) с ходом МИН'а. Здесь s_i – дочерние вершины для вершины s.

Результирующий псевдокод (в виде функции **value(state)** с косвенной рекурсией, возвращающей оценку (ценность, полезность) состояния **state**) для минимаксного поиска представлен ниже:

```
def value(state):
    if state является терминальным: return utility(state)
    if  агент MAX: return max_value(state)
    if  агент MIN: return min_value(state)

def max_value(state):
    v = -∞
    for succ in successor(state): # для всех дочерних состояний
            v=max(v, value(succ))  # рекурсивный вызов value
    return v

def min_value(state):
    v = +∞
    for succ in successor(state):
            v=min(v, value(succ))  # рекурсивный вызов value
    return v
```

Минимаксный поиск за счет рекурсивного погружения ведет себя подобно поиску в глубину, вычисляя оценки состояний в том же порядке, что и DFS. Поэтому временная сложность алгоритма $O(b^m)$, где m — максимальная глубина дерева поиска.

4.2.3. Игры с несколькими игроками

Многие игры допускают наличие более 2-х игроков. Рассмотрим, как можно распространить идею минимаксного поиска на игры с несколькими игроками (агентами).

Для этого можно заменить единственное значение оценки для каждого узла вектором значений оценок [3]. Например, в игре с тремя игроками A, B и C (рисунок 4.2) с каждым узлом ассоциируется вектор (V_A , V_B , V_C) с тремя оценками. Этот вектор задает полезность состояния с точки зрения каждого игрока.

Для пояснения вычисления динамических оценок в нетерминальных узлах рассмотрим узел, обозначенный на рисунке 4.2, как X. В этом состоянии ход выбирает игрок C, анализируя векторы оценок дочерних состояний: (1, 2, 6) и (4, 2, 3). Поскольку для игрока C состояние с $V_C = 6$ предпочтительнее, то игрок C в состоянии X выбирает первый ход и состоянию X приписывается вектор оценок (1, 2, 6).

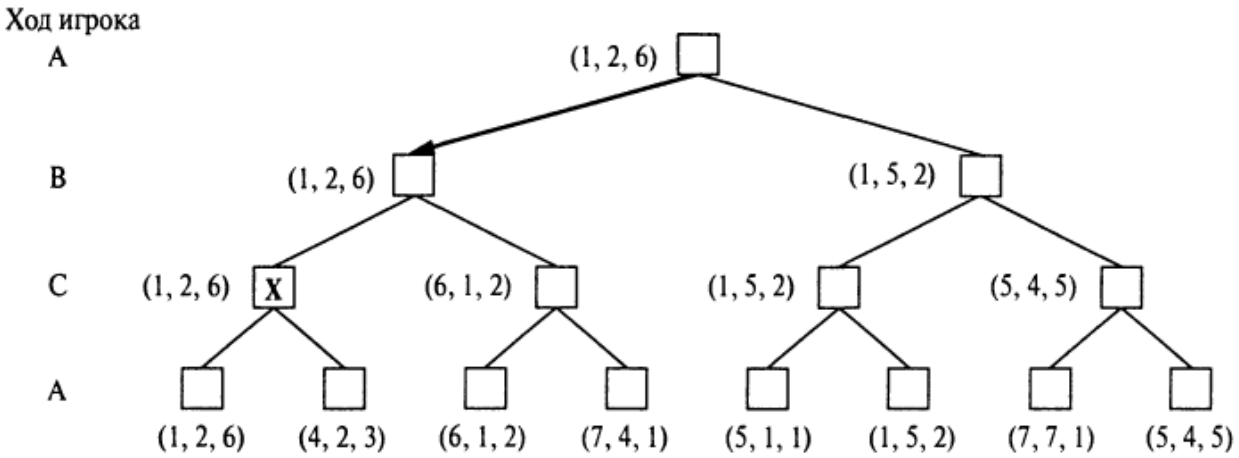


Рисунок 4.2 – Дерево игры с несколькими игроками

Действуя аналогично при всех других состояниях игры, получаем для игрока A в корневом состоянии вектор оценок, равный (1, 2, 6), т.е. игрок A в этом состоянии выберет ход, обозначенный на рисунке 4.2 стрелкой.

4.2.4. Альфа-бета поиск

Минимаксный метод поиска предполагает систематический обход дерева поиска. Альфа-бета поиск позволяет избежать последовательного обхода дерева (рисунок 4.3) за счет отсечения некоторых поддеревьев поиска. Предполагается, что поиск осуществляется сверху вниз и слева направо.

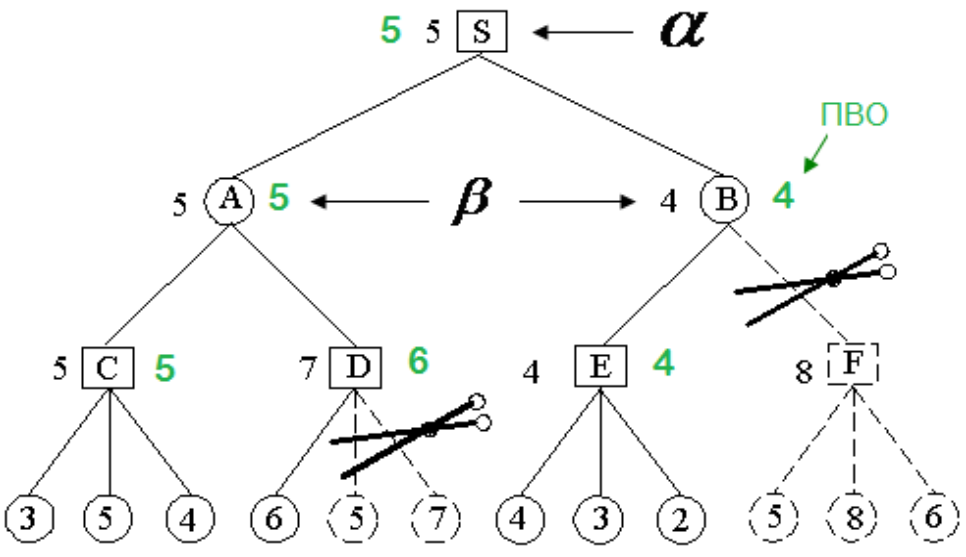


Рисунок 4.3 – Альфа-бета поиск

В альфа-бета методе статическая оценка каждой терминальной вершины вычисляется сразу, как только такая вершина будет построена. Затем полученная оценка распространяется вверх по дереву и с каждой из родительских вершин связывается **предположительно возвращаемая оценка** (ПВО). При этом гарантируется, что для родительской вершины, в которой ход выполняет игрок (ее также

называют ***альфа-вершиной***), уточненная оценка, вычисляемая на последующих этапах, будет не ниже ПВО. Если же в родительской вершине ход выполняет противник (***бета-вершина***), то гарантируется, что последующие оценки будут не выше ПВО. Это позволяет отказаться от построения некоторых вершин и сократить объем поиска. При этом используются следующие правила:

- если ПВО для бета-вершины становится меньше или равной ПВО родительской вершины, вычисленной на предыдущем шаге, то нет необходимости строить дальше поддерево, начинающееся ниже этой бета-вершины;

- если ПВО для альфа-вершины становится больше или равной ПВО родительской вершины, вычисленной на предыдущем шаге, то нет необходимости строить дальше поддерево, начинающееся ниже этой альфа-вершины;

Первое правило соответствует ***альфа-отсечению***, а второе – ***бета-отсечению***. Для дерева поиска, изображенного на рисунке 4.2, ситуация альфа-отсечения имеет место при вычислении ПВО вершины *D* ([*V(D)*=6] > *V(A)*=5]), а ситуация бета-отсечения – для вершины *B* ([*V(B)*=4] < [*V(S)*=5]).

Псевдокод альфа-бета поиска аналогичен минимаксному поиску, но требует переопределения функций, вычисляющих минимальные и максимальные оценки:

```
def max_value(state, α, β):
    v = -∞
    for succ in successor(state):
        v=max(v, value(succ, α, β))
        if v >= β: return v                # альфа отсечение
        α=max(α, v)
    return v

def min_value(state, α, β):
    v = +∞
    for succ in successor(state):
        v=min(v, value(succ, α, β))
        if v <= α: return v                # бета отсечение
        β =min(β, v)
    return v
```

Обратите внимание, что в функциях предусмотрен ранний выход из циклов для случаев бета и альфа отсечений, т.е. полный анализ всех состояний приемников может не выполняться.

Результаты применения альфа-бета поиска зависят от порядка, в котором строятся дочерние вершины. Альфа-бета поиск позволяет увеличить глубину дерева поиска примерно в два раза по сравнению минимаксным алгоритмом, что приводит к более сильной игре [1, 2]. Оценка временной сложности алгоритма альфа-бета поиска соответствует $O(b^{m/2})$.

4.2.5. Функции оценки

В ходе минимаксного поиска процесс генерации ходов останавливают в узлах (состояниях), расположенных на некоторой выбранной глубине. Полезность

этих состояний определяют с помощью тщательно выбранной **функции оценки** (evaluation function), которая дает приближенное значение полезности этих состояний. Чаще всего функция оценки *eval(s)* представляет собой линейную комбинацию признаковых функций $f_i(s)$:

$$eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s),$$

где каждая функция $f_i(s)$ вычисляет некоторую характеристику (признак) состояния s, и каждой характеристике назначается соответствующий вес w_i. Например, в игре в шашки мы могли бы построить оценочную функцию с 4-мя характеристиками: количество пешек у агента, количество королев у агента, количество пешек у противника и количество королев у противника. Структура оценочной функции может быть совершенно произвольной, и она не обязательно должна быть линейной.

4.2.6. Expectimax

Минимаксный поиск бывает чрезмерно пессимистичным в ситуациях, когда оптимальные ответы противника не гарантируются. Для организации поиска в указанных условиях разработан метод, известный как Expectimax.

Expectimax вводит в дерево игры **узлы жеребьевки** (chance node), вместо узлов в которых выбирается минимальная оценка. При этом в узлах жеребьевки вычисляется ожидаемая оценка в виде взвешенного среднего значения. Правило определения ожидаемых значений оценок для узлов жеребьевки выглядит следующим образом:

$$V(s) = \sum_i p(s_i \mid s)\, V(s_i), \qquad (4.1)$$

где $p(s_i|s)$ – условная вероятность того, что данное недетерминированное действие с индексом i приведет к переходу из состояния s в s_i. Суммирование в (4.1) выполняется по всем индексам i.

Минимакс - это частный случай Expectimax: узлы, возвращающие минимальную оценку - это узлы, которые присваивают вероятность 1 своему дочернему узлу с наименьшим значением и вероятность 0 всем другим дочерним узлам. Поэтому псевдокод метода Expectimax очень похож на минимакс, за исключением необходимых изменений, связанных с вычислениями ожидаемой оценки полезности вместо минимальной оценки полезности:

```
def value(state):
    if state является терминальным: return utility(state)
    if  агент MAX: return max_value(state)
    if  агент EXP: return exp_value(state)

def max_value(state):                  # максимальная оценка
    v = -∞
    for succ in successor(state):
          v=max(v, value(succ))
    return v
```

```
def exp_value(state):                    # ожидаемая оценка
    v = 0
    for succ in successor(state):
            p=probability(succ)
            v+=p* value(succ))
    return v
```

Временная сложность Expectimax пропорциональна $O(b^m n^m)$, где n — количество вариантов выпадения жребия.

4.3. Задания для выполнения

Задание 1. Рефлекторный агент ReflexAgent

Усовершенствуйте поведение рефлекторного агента **ReflexAgent** в **multiAgents.py**, чтобы он мог играть “достойно”. Предоставленный код рефлекторного агента содержит несколько полезных методов, которые запрашивают информацию у класса **GameState**. Эффективный рефлекторный агент должен учитывать, как расположение пищевых гранул, так и местонахождение призраков. Усовершенствованный агент должен легко съесть все гранулы на поле игры **testClassic**:

```
python pacman.py -p ReflexAgent -l testClassic
```

Проверьте работу рефлекторного агента на поле **mediumClassic** по умолчанию с одним или двумя призраками (и отключением анимации для ускорения отображения):

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1

python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

Как ведет себя ваш агент? Скорее всего, он будет часто погибать при игре с двумя призраками (на доске по умолчанию), если ваша оценочная функция недостаточно хороша.

Подсказка. Помните, что логический массив **newFood** можно преобразовать в список с координатами пищевых гранул методом **asList()**.

Подсказка. В качестве принципа построения функции оценки попробуйте использовать обратные значения расстояний от Пакмана до пищевых гранул и призраков, а не только сами значения этих расстояний.

Примечание. Функция оценки для рефлекторного агента, которую вы напишете для этого задания, оценивает пары состояние-действие; для других заданий функция оценки будет оценивать только состояния.

Примечание. Будет полезно просмотреть внутреннее содержимое различных объектов для отладки программы. Это можно сделать, распечатав строковые

представления объектов. Например, можно распечатать **newGhostStates** с помощью **print(newGhostStates).**

Опции. Поведение призраков в данном случае является случайным; можно поиграть с более умными направленными призраками, используя опцию **-g DirectionalGhost**. Если случайность не позволяет оценить, улучшается ли ваш агент, то можно использовать опцию **-f** для запуска с фиксированным начальным случайным значением (одинаковый начальный случайный выбор в каждой игре). Можно также запустить несколько игр подряд с опцией **-n.** Отключите при этом графику с помощью опции **-q**, чтобы играть быстрее.

Автооценивание. В ходе оценивания ваш агент запускается для игры на поле **openClassic** 10 раз. Вы получите 0 баллов, если ваш агент просрочит время ожидания или никогда не выиграет. Вы получите 1 очко, если ваш агент выиграет не менее 5 раз, или 2 очка, если ваш агент выиграет все 10 игр. Вы получите дополнительный 1 балл, если средний балл вашего агента больше 500, или 2 балла, если он больше 1000. Вы можете оценить своего агента для этих условий командой

```
python autograder.py -q q1
```

Для работы без графики используйте команду

```
python autograder.py -q q1 --no-graphics
```

Не тратьте слишком много времени на усовершенствование решения этого задания, поскольку основные задания лабораторной работы впереди.

Задание 2. Минимаксный поиск

Необходимо реализовать минимаксного агента, для которого имеется «заглушка» в виде класса **MinimaxAgent** (в файле **multiAgents.py**). Минимаксный агент должен работать с любым количеством призраков, поэтому вам придется написать алгоритм, который будет немного более общим, чем тот, который представлен в разделе 4.2.2. В этом случае минимаксное дерево поиска будет иметь несколько минимальных слоев (по одному для каждого призрака) для каждого максимального слоя. Реализуемый код агента также должен будет выполнять поиск до заданной глубины поддерева игры. Оценки в концевых вершинах минимаксного дерева вычисляются с помощью функции **self.evaluationFunction,** которая по умолчанию соответствует реализованной функции **scoreEvaluationFunction.** Класс **MinimaxAgent** наследует свойства суперкласса **MultiAgentSearchAgent**, который предоставляет доступ к функциям **self.depth** и **self.evaluationFunction.** Убедитесь, что ваш код использует эти функции, где это уместно, поскольку именно эти функции вызываются путем обработки соответствующих параметров командной строки.

Обратите внимание на то, что один слой дерева поиска соответствует одному действию Pacman и последовательным действиям всех агентов-призраков.

Автооцениватель определит, исследует ли ваш агент правильное количество игровых состояний. Это единственный надежный способ обнаружить некоторые очень тонкие ошибки в реализациях минимакса. В результате автооцениватель будет очень требователен к числу вызовов метода **GameState.generateSuccessor**. Если

метод будет вызываться больше или меньше необходимого количества раз, автооцениватель отметит это. Чтобы протестировать и отладить код задания, выполните команду:

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

Результаты тестирования покажут, как ведет себя ваш алгоритм на нескольких небольших деревьях, а также в целом в игре Pacman. Чтобы запустить код без графики, используйте команду:

```
python autograder.py -q q2 --no-graphics
```

Подсказки и замечания:

- Реализуйте алгоритм рекурсивно, используя вспомогательные функции;
- Правильная реализация минимакса приведет к тому, что Pacman будет проигрывать игру в некоторых тестах. Это не станет проблемой при тестировании: это правильное поведение агента, он пройдет тесты;
- Функция оценки для этого задания уже написана (**self.evaluationFunction**). Вы не должны изменять эту функцию, но обратите внимание, что теперь мы оцениваем состояния, а не действия, как это было с рефлекторным агентом. Планирующие агенты оценивают будущие состояния, тогда как рефлекторные агенты оценивают действия, исходя из текущего состояния;
- Минимаксные значения начального состояния для игры на поле **MinimaxClassic** равны 9, 8, 7, -492 для глубин 1, 2, 3 и 4, соответственно. Обратите внимание, что ваш минимаксный агент часто будет выигрывать (665 игр из 1000), несмотря на пессимистичный прогноз для минимакса глубины 4

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

- Pacman всегда является агентом 0, и агенты совершают действия в порядке увеличения индекса агента;
- Все состояния в случае минимаксного поиска должны относиться к типу **GameStates** и передаваться в **getAction** или генерироваться с помощью **GameState.generateSuccessor**;
- На больших игровых полях, таких как **openClassic** и **mediumClassic** (по умолчанию), вы обнаружите, что минимаксный агент устойчив к умиранию, но плохо ведет себя в отношении выигрыша. Он часто суетится, не добиваясь прогресса. Он может даже метаться рядом с гранулой, не съев ее, потому что не знает, куда бы он пошел после того, как съест гранулу. Не волнуйтесь, если вы заметите такое поведение, в задании 5 эти проблемы будут устранены;
- Когда Пакман считает, что его смерть неизбежна, он постарается завершить игру как можно скорее из-за наличия штрафа за жизнь. Иногда такое поведение ошибочно при случайных перемещениях призраков, но минимаксные агенты всегда исходят из худшего:

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

Убедитесь, что вы понимаете, почему Pacman в этом случае нападает на ближайшее привидение.

Задание 3. Альфа-бета отсечение

Необходимо реализовать программу агента в классе **AlphaBetaAgent**, который использует альфа-бета отсечение для более эффективного обследования минимаксного дерева. Ваш алгоритм должен быть более общим, чем псевдокод, рассмотренный в разделе 4.2.4. Суть задания состоит в том, чтобы расширить логику альфа-бета отсечения на несколько минимизирующих агентов.

Вы должны увидеть ускорение работы (возможно альфа-бета отсечение с глубиной 3 будет работать так же быстро, как минимакс с глубиной 2). В идеале, при глубине 3 на игровом поле **smallClassic** игра должна выполняться со сокростью несколько секунд на один ход или быстрее.

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

Минимаксные значения начального состояния при игре на поле **minimaxClassic** равны 9, 8, 7 и -492 для глубин 1, 2, 3 и 4, соответственно.

Оценивание: т.к. проверяется, исследует ли ваш код требуемое количество состояний, то важно, чтобы вы выполняли альфа-бета отсечение без изменения порядка дочерних элементов. Иными словами, состояния-преемники всегда должны обрабатываться в порядке, возвращаемом **GameState.getLegalActions**. Также не вызывайте **GameState.generateSuccesso**r чаще, чем необходимо.

Вы ***не должны выполнять отсечение при равенстве оценок***, чтобы соответствовать набору состояний, который исследуется автооценивателем. Для реализации этого задания используйте код из раздела 4.2.4.

Для проверки вашего кода выполните команду

```
python autograder.py -q q3
```

Результаты покажут, как ведет себя ваш алгоритм на нескольких небольших деревьях, а также в целом в игре pacman. Чтобы запустить код без графики, используйте команду:

```
python autograder.py -q q3 --no-graphics
```

Правильная реализация альфа-бета отсечения приводит к тому, что Pacman будет проигрывать на некоторых тестах. Это не создаст проблем при автооценивании: так как это правильное поведение. Ваш агент пройдет тесты.

Задание 4. Expectimax

Минимаксный и альфа-бета поиски предполагают, что игра осуществляется с противником, который принимает оптимальные решения. Это не всегда так. В этом задании необходимо реализовать класс **ExpectimaxAgent**, который предназначен для моделирования вероятностного поведения агентов, которые могут совершать неоптимальный выбор.

Чтобы отладать свою реализацию на небольших игровых деревьях, используя команду:

```
python autograder.py -q q4
```

Если ваш алгоритм будет работать на небольших деревьях поиска, то он будет успешен и при игре в Pacman.

Случайные призраки, конечно, не являются оптимальными минимаксными агентами, и поэтому применение в этой ситуации минимаксного поиска не является подходящим. Вместо того, чтобы выбирать минимальную оценку для состояний с действиями призраков, **ExpectimaxAgent** выбирает ожидаемую оценку. Чтобы упростить код, предполагается, что в этом случае призрак выбирает одно из своих действий, возвращаемых **getLegalActions,** равновероятно.

Чтобы увидеть, как **ExpectimaxAgent** ведет себя при игре в Pacman, выполните команду:

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
```

Теперь вы должны наблюдать иное поведение агента в непосредственной близости к призракам. В частности, если Пакман понимает, что может оказаться в ловушке, но может убежать, чтобы схватить еще несколько гранул еды, он, по крайней мере, попытается это сделать. Изучите результаты этих двух сценариев:

```
python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

Вы должны обнаружить, что теперь **ExpectimaxAgent** выигрывает примерно в половине случаев, в то время как ваш **AlphaBetaAgent** всегда проигрывает. Убедитесь, что вы понимаете, почему поведение этого агента отличается от минимаксного случая.

Правильная реализация Expectimax приведет к тому, что Pacman будет проигрывать некоторые тесты. Это не создаст проблем при автооценивании. Ваш агент пройдет тесты.

Задание 5. Функция оценки

Реализуйте лучшую оценочную функцию для игры Pacman в предоставленном шаблоне функции **betterEvaluationFunction**. Функция оценки должна оценивать состояния, а не действия, как это делала функция оценки рефлекторного агента. При поиске до глубины 2 ваша функция оценки должна обеспечивать выигрыш на поле **smallClassic** с одним случайным призраком более чем в половине случаев и по-прежнему работать с разумной скоростью (чтобы получить хорошую оценку за это задание, Pacman должен набирать в среднем около 1000 очков, когда он выигрывает).

Оценивание: в ходе автооценивания агент запускается 10 раз на поле **smallClassic**. При этом вы получаете следующие баллы:

Если вы выиграете хотя бы один раз без тайм-аута автооценивателя, вы получите 1 балл. Любой агент, не удовлетворяющий этим критериям, получит 0 баллов;

+1 за победу не менее 5 раз, +2 за победу в 10 попытках;

+1 для среднего количества очков не менее 500, +2 за среднее количество очков не менее 1000 (включая очки в проигранных играх);

+1, если ваши игры с автооценивателем в среднем требуют менее 30 секунд при запуске с параметром **--no-graphics**;

Дополнительные баллы за среднее количество очков и время вычислений будут начислены только в том случае, если вы выиграете не менее 5 раз.

Пожалуйста, не копируйте файлы из предыдущих лабораторных работ, так как они не пройдут автооценивание на поле Gradescope.

Вы можете оценить своего агента, выполнив команду

```
python autograder.py -q q5
```

Для выполнения с отключенной графикой используйте команду:

```
python autograder.py -q q5 --no-graphics
```

4.4. Порядок выполнения лабораторной работы

4.4.1. Изучить по лекционному материалу и учебным пособиям [1-3] методы состязательного поиска: минимакс, альфа-бета отсечение и Expectimax.

4.4.2. Использовать для выполнения лабораторной работы файлы из архива **МиСИИ_лаб4_2024.zip.** Программный код этой лабораторной работы не сильно изменился по сравнению с работами 2 и 3, но, тем не менее, разверните его в новой папке и не смешивайте с файлами предыдущих лабораторных работ.

4.4.3. В этой лабораторной работе необходимо реализовать агентов для классической версии Pacman, включая призраков. При этом потребуется реализовать как минимаксный поиск, так и expectimax поиск, а также разработать оценочные функции.

Как и ранее набор файлов включает в себя автооцениватель, с помощью которого вы можете оценивать свои решения. Вызов автооценивателя для проверки всех заданий можно выполнить с помощью команды:

python autograder.py

Для проверки решения конкретного задания, например 2-го, автооцениватель можно вызвать с помощью команды:

python autograder.py -q q2

Для запуска конкретного теста используйте команду вида:

python autograder.py -t test_cases/q2/0-small-tree

По умолчанию автооцениватель отображает графику с параметром **–t**. Можно принудительно включить графику с помощью флага **--graphics** или отключить графику с помощью флага **--no-graphics**.

Код этой лабораторной работы содержит следующие файлы:

Файлы для редактирования:	
`multiAgents.py`	Здесь будут размещаться все ваши мультиагентные поисковые агенты, которых вы определите.
Файлы, которые необходимо просмотреть	
pacman.py	Основной файл, из которого запускают Pacman. Этот файл описывает тип Pacman GameState, который используется в лабораторных работах.
game.py	Логика, лежащая в основе мира Pacman. Этот файл описывает несколько поддерживающих типов, таких как AgentState, Agent, Direction и Grid.
util.py	Полезные структуры данных для реализации алгоритмов поиска.
Поддерживающие файлы, которые можно игнорировать:	
graphicsDisplay.py	Графика Pacman
graphicsUtils.py	Графические утилиты
textDisplay.py	ASCII графика Pacman
ghostAgents.py	Агенты, управляющие привидениями
keyboardAgents.py	Интерфейс клавиатуры для управления игрой
layout.py	Код для чтения файлов схем и хранения их содержимого
autograder.py	Автооцениватель
testParser.py	Парсер тестов автооценивателя и файлы решений
testClasses.py	Общие классы автооценивателя
test_cases/	Папка, содержащая тесты для кадого из заданий (вопросов)
multiagentTestClasses.py	Специальные тестовые классы автооценивателя для данной лабораторной работы

4.4.4. Сначала поиграйте в классический Pacman, выполнив следующую команду:

python pacman.py

Используйте клавиши со стрелками для перемещения. Запустите предоставленный рефлекторный агент **ReflexAgent** в **multiAgents.py**

python pacman.py -p ReflexAgent

Обратите внимание, что он плохо играет даже в случае простых вариантов игры:

python pacman.py -p ReflexAgent -l testClassic

Изучите код рефлекторного агента (в **multiAgents.py**) и убедитесь, что вы понимаете, как он работает. Для этого код рефлекторного агента снабжен необходимыми комментариями.

4.4.5. Выполните задание 1 - усовершенствуйте поведение рефлекторного агента **ReflexAgent.** Используйте подсказки и советы, указанные в задании 1. После вставки кода усовершенствования агента выполните автооценивание задания 1 и результаты внесите в отчет.

4.4.6. Выполните задание 2 - реализуйте минимаксный поиск с произвольным количеством агентом. В основу построения минимаксного агента положите псевдокод, указанный в разделе 4.2.2, а также общие принципы игры с несколькими игроками, изложенные в разделе 4.2.3. При этом обратите внимание на то, что одному действию (ходу) Пакмана на текущем уровне дерева игры будут соответствовать последовательные действия нескольких агентов-призраков. Пакман выбирает действие с максимальной оценкой, призраки – действие с минимальной оценкой. Глубина дерева поиска увеличивается на 1, каждый раз, когда право совершить действие вновь возвращается к Пакману.

Чтобы организовать такой сценарий мультиагентной игры необходимо расширить пседокод функции **min_value(state)** (см. раздел 4.2.2) на случай нескольких агентов-призраков. Для этого функция **min_value** должна обеспечивать реализацию поиска в глубину путем косвенных рекурсивных вызовов функции **value** с дополнительными входными параметрами, задающими индекс агента **agentIndex** и уровень глубины **depth.** При очередном косвенном рекурсивном вызове **min_value** на одном и том же уровне **depth** дерева поиска параметр **agentIndex** должен увеличиваться на 1 (для передачи хода следующему агенту-призраку). Пример псевдокода:

```
def min_value(state, depth, agentIndex):
        …
    v = +∞
    # Для всех допустимых действий агента с номером agentIndex
    for action in gameState.getLegalActions(agentIndex):
        if agentIndex == NumberOfAgents()-1:
            v = min(v, value(successor(state,agentIndex, action),depth + 1, 0))
        else:
            v = min(v, value(successor(state, agentIndex, action), depth, agentIndex+1))
      return v
```

Когда все агенты-призраки (число которых определяется значением **NumberOfAgents)** выполнят свои действия, ход передается Пакману (индекс агента 0) и глубина поиска увеличивается на 1.

После отладки программы выполните автооценивание задания 2 и результаты внесите в отчет.

4.4.7. Выполните задание 3 – реализуйте альфа-бета поиск. Для этого используйте код минимаксного агента, который был реализован в задании 2 и дополните его альфа- и бета-отсечениями, рассмотренными в разделе 4.2.4.

После отладки программы выполните автооценивание задания 3 и результаты внесите в отчет.

4.4.8. Выполните задание 4 – реализуйте класс **ExpectimaxAgent**, который моделирует вероятностное поведение агентов. Для этого используйте код минимаксного агента, в котором замените код функции **min_value() (**см. п.4.4.6) на код функции **exp_value()** . Псевдокод функции приведен в разделе 4.2.6. При реализации функции **exp_value()** используйте тот же перечень параметров, который использует функция **min_value().**

При реализации агента **ExpectimaxAgent** выбор допустимых действий в соответствии с заданием 4 должен быть равновероятным в этом случае формула (4.1) сводится к формуле обычного среднего значения, т.к. $p(s_i \mid s) = 1/N$, N – число состояний-приемников для состояния s.

Код функции **exp_value()** легко получается путем замены в функции **min_value()** строк

```
v = min(v, value(successor(state,agentIndex, action),…)
```

на

```
v += value(successor(agentIndex, action), …)
```

В этом случае в переменной **v** будет накапливаться сумма в соответствии с формулой (4.1). Для вычисления средней оценки необходимо определить количество допустимых действий и поделить указанную сумму **v** на количество действий:

```
v/len(gameState.getLegalActions(agentIndex))
```

После отладки программы агента **ExpectimaxAgent** выполните автооценивание задания 4 и результаты внесите в отчет.

4.4.9. Задание 5 сводится к определению улучшенной версии функции оценки **betterEvaluationFunction** состояния игры. Вы должны придумать эффективные правила вычисления оценок состояний, которые обеспечат более разумное поведение Пакмана. Руководствуйтесь рекомендациями, изложенными в разделе 4.2.5. Начните с анализа недостатков функции **evaluationFunction(self, currentGameState, action)** , которую Вы определяли в задании 1. Теперь функция должна оценивать перспективность состояний. В этом смысле определяемая функция оценки соответствует эвристической функции, применяемой в A*-алгоритме. Перспективность состояний с точки зрения победы Пакмана может быть выражена,

например, в виде дополнительных баллов, начисляемых за ход в сторону расположения ближайшей пищевой гранулы или за ход, в сторону испуганного призрака.

После отладки функции выполните автооценивание задания 5 и результаты внесите в отчет.

4.5. Содержание отчета

Цель работы, краткий обзор методов поиска решений задач в игровых программах, описание свойств методов, код реализованных агентов и функций с комментариями в соответствии с заданиями 1-5, результаты игры на разных полях игры, их анализ, результаты автооценивания заданий, выводы по проведенным экспериментам с разными состязательными агентами.

4.6. Контрольные вопросы

4.6.1 Объясните понятия: детерминированные игры с нулевой суммой, состязательный поиск.

4.6.2. Объясните следующие понятия минимаксного поиска: дерево поиска, статические оценки, динамические оценки, функция полезности.

4.6.3. Объясните на примере принцип минимаксного поиска и запишите формальные выражения, используемые для распространения оценок по дереву поиска.

4.6.4. Разработайте пример дерева игры в крестики-нолики и предложите способ вычисления статических оценок.

4.6.5. Запишите псевдокод, определяющий основные функции минимаксного рекурсивного поиска и объясните его на примере.

4.6.6. Объясните особенности игр с несколькими игроками, приведите пример соответствующего дерева игры и объясните механизм распространения оценок состояний.

4.6.7 Объясните на примере принцип альфа-бета поиска, сформулируйте правила альфа- и бета-отсечений.

4.6.8. Запишите псевдокод, определяющий основные функции альфа-бета поиска и объясните их работу на примере.

4.6.9. Что такое функция оценки? В какой математической форме её обычно определяют?

4.6.10. Почему минимаксный принцип построения агентов, функционирующих в средах с элементами случайности, недостаточно эффективен?

4.6.11. Сформулируйте метод поиска Expectimax, что такое узлы жеребьевки, как в этих узлах вычисляется ожидаемая оценка?

4.6.12. Запишите псевдокод, определяющий основные функции Expectimax поиска и объясните их работу на примере.

5. ЛАБОРАТОРНАЯ РАБОТА № 5 «ИССЛЕДОВАНИЕ СЕТЕЙ БАЙЕСА И СММ»

5.1. Цель работы

Исследование методов точного и приближенного вероятностного вывода с использованием сетей Байеса и скрытых марковских моделей, приобретение навыков программирования интеллектуальных агентов, знания которых представляются условными высказываниями с определенной степенью уверенности.

5.2. Краткие теоретические сведения

5.2.1. Неопределенность и степени уверенности высказываний

Интеллектуальные агенты почти никогда не имеют доступа ко всей информации о среде функционирования. Поэтому они действуют в условиях неопределенности. При функционировании агента в среде с неопределенностями знания агента в лучшем случае позволяют сформировать относящиеся к делу высказывания только с определенной **степенью уверенности/убежденности** (degree of belief) [1, 2, 3]. Основным инструментом, применяемым для обработки степеней уверенности таких высказываний и осуществления вывода, является теория вероятностей, в которой каждому высказыванию присваивается числовое значение степени уверенности в диапазоне от 0 до 1.

Напомним основные понятия и соотношения, используемые при построении вероятностных моделей [3, 9].

5.2.2. Совместные и условные вероятности

Совместное распределение множества случайных переменных $X_1, X_2,\ldots, X_n$ определяет вероятность каждого возможного присвоения (или исхода):

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n) = P(x_1, x_2, \ldots x_n). \qquad (5.1)$$

Свойства совместного распределения:

$$P(x_1, x_2, \ldots x_n) \geq 0,$$

$$\sum_{(x_1, x_2, \ldots x_n)} P(x_1, x_2, \ldots x_n) = 1$$

Событие – это подмножество E некоторых возможных исходов. Вероятность исходов события E равна:

$$P(E) = \sum_{(x_1 \ldots x_n) \in E} P(x_1 \ldots x_n)$$

Одна из задач, которая часто встречается, состоит в том, чтобы извлечь из совместного распределения некоторое частное распределение по нескольким или одной переменной. Например,

$$P(X_1 = x_1) = \sum_{x_2} P(X_1 = x_1, X_2 = x_2),$$

складывая вероятности по переменной X_2 при фиксированных значениях X_1, получим распределение по одной переменной $P(X_1)$ (**маргинальное распределение**). Такой процесс называется **маргинализацией,** или исключением из суммы, поскольку из суммы вероятностей исключаются прочие переменные, кроме X_1.

Условная вероятность $P(x|y)$, т.е. вероятность x при известном y, определяется на основе совместной вероятности следующим образом:

$$P(x|y) = \frac{P(x,y)}{P(y)}. \tag{5.2}$$

Иногда требуется по условной вероятности определить совместную вероятность. Тогда используют **правило произведения**

$$P(y)P(x|y) = P(x,y). \tag{5.3}$$

В общем случае можно представить любую совместную вероятность как последовательное произведение условных вероятностей (**цепочное правило**):

$$P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2),$$

$$P(x_1, x_2, \ldots x_n) = \prod_i P(x_i|x_1 \ldots x_{i-1}) \tag{5.4}$$

Совместную вероятность 2-х переменных можно представить в виде:

$$P(x,y) = P(x|y)P(y) = P(y|x)P(x)$$

Выполнив деление, получим **правило Байеса**:

$$P(x|y) = \frac{P(y|x)}{P(y)}P(x) \tag{5.5}$$

Правило позволяет выразить одну условную вероятность через другую, которую бывает вычислить проще.

5.2.3. Независимость и условная независимость

Две случайные переменные (абсолютно) **независимы**, если [3, 9]:

$$\forall x, y : P(x,y) = P(x)P(y). \tag{5.6}$$

Совместное распределение $P(X,Y)$ независимых переменных представляется (факторизуется**)** в виде произведения двух более простых распределений. Независимость случайных переменых X и Y обозначают в виде $X \perp\!\!\!\perp Y$.

Общее определение **условной независимости** двух переменных *X* и *Y*, если дана третья переменная *Z*: *X условно не зависит от Y при заданном Z, если и только если:*

$$\forall x, y, z : P(x, y|z) = P(x|z)P(y|z) \text{, т.е. } X \perp\!\!\!\perp Y|Z \quad (5.7)$$

или, эквивалентно, если и только если

$$\forall x, y, z : P(x|z, y) = P(x|z). \quad (5.8)$$

Разработка методов декомпозиции крупных предметных областей на слабо связанные подмножества предметных переменных с помощью свойства условной независимости стало одним из наиболее важных достижений в истории искусственного интеллекта [9].

5.2.4. Байесовские сети

Байесовская сеть представляет совместную вероятность множества *n* случайных переменных X_1, X_2,…, X_n в форме направленного ациклического графа. Каждая вершина графа представляется случайной переменной, с которой связана **таблица условных вероятностей** (CPT – conditional probability table), содержащая вероятности переменной с учетом её условных родительских переменных в графе. Ребра графа обозначают взаимодействия переменных. Однако, важно помнить, что они не означают причинные связи [3, 9].

Произвольная вершина *X* сети Байеса описывается локальным условным распределением:

$$P(X \mid A_1, A_2, \ldots A_n) , \quad (5.9)$$

где $A_1, A_2, \ldots A_n$ – родительские (*parents*) переменные (вершины) для *X*. CPT переменной содержит *n*+2 столбцов: один для хранения значений каждой из *n* родитедьских переменных, один для *X* и один для значения условной вероятности *X* при заданом значении *Y*.

При известных CPT для каждой вершины сети Байеса, можно вычислить вероятности заданных совместных присваиваний всех переменных сети на основе правила:

$$P(X_1, X_2, \ldots X_n) = \prod_{i=1}^{n} X_i \mid parents(X_i) \quad (5.10)$$

Утверждения условной независимости, кодируемых сетью Байеса, можно проверить (верифицировать) путём анализа структуры сети, используя критерий, называемый **D-разделенностью** [2, 3, 9].

Отдельные элементы произведения (5.10) в ходе поиска ответов на запросы называют **факторами**. **Фактор** хранит таблицу “вероятностей”, хотя сумма записей в таблице не обязательно равна 1. Фактор в общем случае имеет форму $f(X_1,\ldots,X_m, y_1,\ldots,y_n \mid Z_1,\ldots,Z_p, w_1,\ldots,w_q)$. Напомним, что строчные буквы обозначают переменные, которые уже получили значение. Для каждого возможного значения

переменных X_i и Z_j в таблице вероятностей фактора хранится одно число. Переменные Z_j и w_k называются **условными**, а переменные X_i и y_l — **безусловными**. В сети Байеса фактор, представляемый **таблицей условных вероятностей** (CPT), обладает двумя свойствам: 1) элементы в сумме должны давать 1 для каждого назначения условных переменных; 2) фактор имеет ровно одну безусловную переменную.

Вероятностный вывод предполагает вычисление вероятностей переменных запроса через вероятности других известных переменных. Известны алгоритмы **точного вероятностного вывода** (например, вывод путем перечисления или путем исключения переменных), которые характеризуются большой вычислительной сложностью. Точный вероятностный вывод при большом числе переменных часто неосуществим. Поэтому были разработаны **методы приближенного вероятностного вывода**, основанные на формировании случайных выборок из распределений. Вероятностный вывод на основе выборок осуществляется быстрее, чем вычисление ответа на запрос, например, путем исключения переменных. Точность вывода зависит от количества формируемых выборок.

5.2.5. Вывод путем перечислений

Располагая совместным распределением, мы можем вычислить любое желаемое распределение вероятностей, представляемое **запросом** $P(Q_1,\ldots,Q_m \mid e_1,\ldots,e_n)$, где Q_i –переменные запроса, e_1 – переменные свидетельства, которые являются наблюдаемыми переменными, значения которых известны.

Вывод путем перечислений (**Inference By Enumeration**) реализуется следующим алгоритом:

1. Выбираем все строки таблицы совместного распределения, которые содержат наблюдаемые переменные свидетельств;
2. Суммируем строки, содержащие скрытые переменные (исключаем). Скрытые переменные – это те переменные, которые присутствуют в общем совместном распределении, но отсутсвуют в запросе;
3. Нормализуем таблицу так, чтобы она представляла собой распределение вероятностей (т. е. сумма значений равнялась 1).

5.2.6. Вывод путем исключения переменных

Альтернативный подход заключается в исключении скрытых переменных по одной. Чтобы исключить скрытую переменную Y (не входит в запрос) необходимо:

1. Объединить (перемножить) все **факторы**, включающие Y, например: $P(X|Y)\ P(Y)=P(X, Y)$;

2. Выполнить суммирование по Y (исключить Y): $P(X) = \sum_Y P(X, Y)$.

Важно отметить, что вывод путем исключения переменных улучшает вывод путем перечисления только в том случае, если размер наибольшего фактора ограничен разумным значением.

5.2.7. Приближенный вывод в сетях Байеса: формирование выборки

Иной подход к построению вероятностных выводов заключается в неявном вычислении вероятностей запроса путем простого подсчета выборочных значений. Это не даёт точного решения, но часто решение бывает достаточно хорошим, особенно если учитывать огромную экономию в вычислениях.

Алгоритм формирования выборки из заданного дискретного распределения можно представить в виде 2-х шагов:

1. Получить случайное число *u* из равномерного распределения в интервале [0, 1). Например, можно использовать функцию **random()** языка Пайтон;
2. Преобразовать это значение *u* в выборочное значение дискретной случайной переменной с учетом заданного распределения, связав *u* с некоторым диапазоном, ширина которого равна задаваемой распределением вероятности.

Пример. Пусть задано распределение цветов (таблица 5.1), которое в памяти можно хранить в виде словаря с набором пар **{C: P(C)}**.

Таблица 5.1. – Распределение цветов

C	P(C)
red	0.6
green	0.1
blue	0.3

Введем диапазоны, ширина которых равна вероятностям цветов:

$$0 \leq u < 0.6, \rightarrow C = red$$
$$0.6 \leq u < 0.7, \rightarrow C = green$$
$$0.7 \leq u < 1, \rightarrow C = blue$$

Тогда, если **random()** возвращает $u = 0.83$, то выборное значение C = blue. После формирования большого числа выборок можно получить набор выборочных значений с вероятностями, сходящимися к заданному распределению.

5.2.8. Марковские Модели

Сети Байеса представляют собой универсальную структуру, используемую для компактного представления отношений между случайными величинами. Марковскую модель можно рассматривать как аналог байесовской сети в виде внутренне связанной структуры бесконечной длины, зависящей от времени.

Рассмотрим пример моделирования погодных условий с помощью марковской модели. Определим W_i как случайную переменную, представляющую состояниие погоды в *i*-ый день. Модель Маркова для примера погоды изображена на рисунке 5.1.

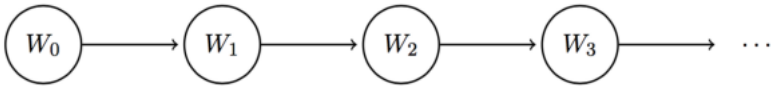


Рисунок 5.1.— Модель Маркова

Начальное состояние в примере марковской модели задано таблицей вероятностей $Pr(W_0)$. Модель перехода из состояния W_i в W_{i+1} задается условным распределением $Pr(W_{i+1}|W_i)$, т.е. погода в момент времени $t = i + 1$ удовлетворяет марковскому свойству (модель без памяти) и не зависит от погоды во все другие моменты времени, кроме $t = i$.

В общем случае на каждом временном шаге в марковских моделях делают следующее **предположение независимости** W_{i+1} _||_ $\{W_0,\ldots, W_{i-1}\}|W_i$. Это позволяет восстановить совместное распределение для первых $n + 1$ переменных с помощью цепочного правила следующим образом:

$$Pr(W_0,W_1,\ldots,W_n) = Pr(W_0)Pr(W_1|\ W_0)Pr(W_2|W_1)\ldots\ Pr(W_n|W_{n-1}). \qquad (5.11)$$

Чтобы определить распределение погодных условий в произвольный день используют алгоритм прямого распространения. В соответствии со свойством маргинализации

$$Pr(W_{i+1}) = \sum_{wi} Pr(w_i,\ W_{i+1}). \qquad (5.12)$$

Применив цепочное правило, получим выражение, определяющее **алгоритм прямого распространения** (mini-forward алгоритм):

$$Pr(W_{i+1}) = \sum_{wi}\ Pr(W_{i+1}|w_i)Pr(w_i). \qquad (5.13)$$

Алгоритм позволяет итеративно вычислять распределение W_{i+1} для произвольно заданного момента времени, начиная с априорного распределения $Pr(W_0)$.

После большого числа шагов мы приходим к **стационарному распределению**, которое слабо зависит от начального распределения. При большом числе шагов на результат оказывает доминирующее влияние переходное распределение.

5.2.9. Скрытые марковские модели

Скрытые марковские модели (СММ) описываются с помощью двух вероятностных процессов: скрытого процесса смены состояний цепи Маркова и наблюдаемых значений свидетельств, формируемых при смене состояний.

В качестве примера на рисунке 5.2 изображена скрытая модель Маркова для моделирования погоды. В отличие от обычной марковской модели, СММ содержит два типа узлов: скрытые узлы W_i , которые являются **переменными состояния** и представляют погоду в i-ый день, и наблюдаемые узлы F_i , которые представляют **переменные, называемые свидетельствами (наблюдениями).** В рассматриваемом примере свидетельства F_i представляют прогноз погоды в i-ый день.

Одна из задач, решаемая с помощью модели СММ и называемая **фильтрацией или мониторингом**, заключается в вычислении апостериорных распределений скрытых переменных состояний в текущий момент времени по значениям всех полученных к этому моменту свидетельств.

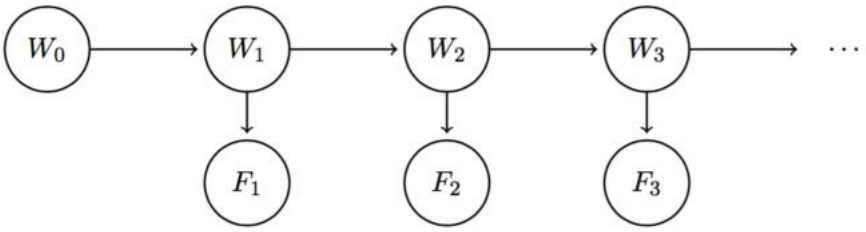


Рисунок 5.2. – Скрытая модель Маркова

СММ подразумевает такие же **условные отношения независимости**, как и для стандартной марковской модели, с дополнительным набором отношений для переменных свидетельств:

$$F_1 \_||\_ W_0|W_1 ,$$
$$\forall i = 2,\ldots,n \quad W_i \_||\_ \{ W_0,\ldots, W_{i-2}; F_1,\ldots., F_{i-1}\} \mid W_{i-1},$$
$$\forall i = 2,\ldots,n \quad F_i \_||\_ \{ W_0,\ldots, W_{i-1}; F_1,\ldots, F_{i-1} \} \mid W_i. \qquad (5.14)$$

Как и в марковских моделях, в СММ предполагается, что переходная модель $Pr(W_{i+1}|W_i)$ является стационарной. СММ делают дополнительное упрощающее предположение, что **модель восприятия (модель наблюдения, сенсорная модель)** $Pr(F_i|W_i)$ также является стационарной. *Следовательно, любая СММ может быть компактно представлена с помощью всего лишь трех таблиц вероятностей: начального распределения, модели перехода и модели наблюдения.*

Рассмотрим **алгоритм прямого распространения для СММ**. Определим *распределение степеней уверенности* (belief distribution) относительно возможных значений состояния W_i по всем свидетельствам $f_1,\ldots, f_i$, поступившим к моменту времени i как:

$$B(W_i) = Pr(W_i| f_1,\ldots, f_i). \qquad (5.15)$$

Аналогично определим через $B'(W_i)$ оценку распределения степеней уверенности (убеждений) в момент времени i по наблюдениям $f_1,\ldots, f_{i-1}$, которые поступили к моменту времени i-1, т.е. оценку $B'(W_i)$ можно рассматривать как *прогноз на один шаг вперед*:

$$B'(W_i) = Pr(W_i| f_1,\ldots, f_{i-1}) \qquad (5.16$$

Найдем соотношение между $B(W_i)$ и $B'(W_{i+1})$. Начнем с определения $B'(W_{i+1})$:

$$B'(W_{i+1}) = Pr(W_{i+1}| f_1,\ldots, f_i)= \sum_{wi} Pr(W_{i+1}, w_i \mid f_1,\ldots, f_i). \qquad (5.17)$$

Перепишем соотношение с использованием цепочного правила (5.4):

$$B'(W_{i+1}) = Pr(W_{i+1} \mid f_1, \ldots, f_i) = \sum_{wi} Pr(W_{i+1}, |w_i , f_1, \ldots, f_i)\, Pr(w_i \mid f_1, \ldots, f_i).$$

Так как $Pr(w_i|f_1, \ldots, f_i) = B(w_i)$ и W_{i+1} _||_ $\{f_1, \ldots, f_i\}|W_i$, то из последнего выражения следует **правило обновления во времени (**Time Elapse Update**),** которое распространяет распределение $B(W_i)$ с помощью модели перехода $Pr(W_{i+1}, |w_i)$ на один шаг вперед во времени и позволяет определить $B'(W_{i+1})$

$$B'(W_{i+1}) = \sum_{wi} Pr(W_{i+1}, |w_i)\, B(w_i). \qquad (5.18)$$

Найдем связь между $B'(W_{i+1})$ и $B(W_{i+1})$. Из правила Байеса (5.5) следует:

$$B(W_{i+1}) = Pr(W_{i+1}|f_1, \ldots, f_{i+1}) = \frac{Pr(W_{i+1}, f_{i+1}|f_1, \ldots, f_i)}{Pr(f_{i+1}|f_1, \ldots, f_i)}$$

Опуская операцию деления на знаменатель (операция нормализации), перепишем выражение с использованием цепочного правила:

$$B(W_{i+1}) \propto Pr(W_{i+1}, f_{i+1}|f_1, \ldots, f_i) = Pr(f_{i+1}|W_{i+1}, f_1, \ldots, f_i) Pr(W_{i+1}|f_1, \ldots, f_i).$$

В соответствии с предположениями условной независимости для СММ и определением $B'(W_{i+1})$ получим **правило обновления** $B'(W_{i+1})$ **на основе наблюдения** (Observation Update) $Pr(f_{i+1}|W_{i+1})$:

$$B(W_{i+1}) \propto Pr(f_{i+1}|W_{i+1}) B'(W_{i+1}). \qquad (5.19)$$

Объединение полученных правил дает итерационный алгоритм, известный как **алгоритм прямого распространения для СММ** (аналог mini-forward алгоритма для обычной марковской модели). Алгоритм включает два отдельных шага:

1. Обновление $B'(W_{i+1})$ по $B(W_i)$ на одном шаге во времени;
2. Обновление $B(W_i)$ на основе наблюдения, т.е. определение $B(W_{i+1})$ по $B'(W_{i+1})$.

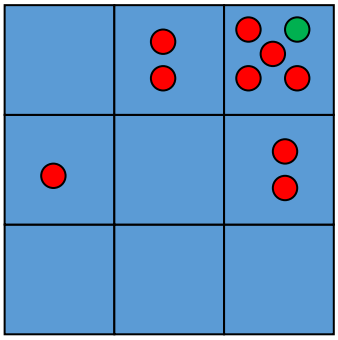
Отметим, что указанный выше трюк с нормализацией может значительно упростить вычисления. Если мы начнем с некоторого начального распределения и будем вычислять распределение степеней уверенности в момент времени *t,* то можно использовать прямой алгоритм для итеративного вычисления $B(W_1), \ldots, B(W_t)$ и выполнять нормализацию только один раз в конце, разделив каждую запись в таблице для $B(W_t)$ на сумму её записей.

5.2.10. Фильтрация частиц

Точный вывод с использованием алгоритма прямого распространения СММ характеризуется большой вычислительной сложностью. В этом случае, аналогично сетям Байеса, используют приближенные методы вывода, основанные на формировании случайных выборок из распределений [3].

Примение к СММ процедур, аналогичных байесовскому сэмплированию (взятию выборок), называется **фильтрацией частиц** и включает в себя моделирование движения набора частиц через граф состояний для аппроксимации распределения вероятностей (доверий) рассматриваемой случайной величины в требуемый момент времени. Частица в этом случает представляет возможное выборочное значение случайной величины. При этом вместо хранения полных таблиц вероятностей, отображающих каждое состояние в вероятность, хранят список из n частиц, в котором каждая частица может находиться в одном из d состояний. Чем больше частиц, тем выше точность аппроксимации.

В качестве примера на рисунке 5.3. изображен **список частиц**, представленных координатами клеток, в которых они расположены. Соответственно, для частиц с координатами (3,3) эмпирическая оценка вероятности появления в списке частиц равна $B(3,3)=5/10=0.5$. Таким образом, по списку частиц можно восстановить эмпирическое распределение случайной переменной.



Список из 10 частиц:
(3,3), (2,3), (3,3), (3,2), (3,3), (3,2), (1,2), (3,3), (3,3),(2,3)

Рисунок 5.3. – Список частиц

Моделирование фильтрации частиц начинается с инициализации частиц. Например, можно выбрать частицы случайным образом из некоторого начального распределения. После того, как выбран исходный список частиц, моделирование принимает форму, аналогичную алгоритму прямого распространения в СММ с чередованием обновления распределений во времени и обновления на основе наблюдения на каждом временном шаге.

Обновление во времени (Time Elapse Update) — обновление выборочного значения каждой частицы в соответствии с моделью переходной вероятности. Для частицы в состоянии t_i выполняется случайная выборка обновленного значения из переходного распределения $Pr(T_{i+1} \mid t_i)$.

Обновление на основе наблюдения (Observation Update). Этот этап немного сложнее. Здесь используется модель сенсора $Pr(F_i|T_i)$ для взвешивания каждой частицы в соответствии с вероятностью, определяемой наблюдаемым свидетельством и состоянием частицы. В частности, частице в состоянии t_i при свидетельстве f_i , поступающим от некоторого сенсора, присваивается вес $Pr(f_i|t_i)$. Алгоритм обновления на основе наблюдений следующий:

1. Рассчитайте веса всех частиц в соответствии с $Pr(f_i|t_i)$;
2. Вычислите суммарный вес каждого состояния;

3. Если сумма всех весов во всех состояниях равна 0, повторно инициализируйте все частицы;
4. Иначе нормализуйте распределение по отношению к суммарному весу и выполните выборки из этого распределения.

Давайте рассмотрим процесс фильтрации частиц (обновление во времени и обновление на основе наблюдения) на примере моделирования погоды. Пусть, например, имеется список из 10 частиц со следующими значениями температуры [15,12,12,10,18,14,12,11,11,10] из дипазона [10,20]. Соответственно, подсчитав количество различных состояний частиц и поделив эти значения на общее число частиц, получим эмпирическое распределение температуры в момент времени *i:*

T_i	10	11	12	13	14	15	16	17	18	19	20
$B(T_i)$	0.2	0.2	0.3	0	0.1	0.1	0	0	0.1	0	0

Определим модель перехода, используя температуру как случайную переменную, зависящую от времени. Будем полагать, что для определенного состояния температура может либо оставаться прежней, либо измениться на один градус в диапазоне [10, 20]. При этом пусть вероятность перехода в следующий момент времени к значению, которое ближе к 15, составляет 0.8, а остальные результирующие состояния равномерно делят оставшиеся 0.2 вероятности между собой.

Чтобы выполнить обновление во времени для первой частицы из списка частиц (T_i = 15), воспользуемся выбранной моделью перехода:

T_{i+1}	14	15	16
$Pr(T_{i+1} \mid T_i = 15)$	0.1	0.8	0.1

Для формирования выборки для частицы в состоянии T_i = 15 воспользуемся алгоритмом сэмплирования, описанным в разделе 5.2.7, в соответствии с которым просто генерируем случайное число в диапазоне [0, 1) и смотрим, в какой диапазон оно попадает. Например, если случайное число равно r = 0.467, то частица с T_i = 15 попадает в диапазон $0{,}1 \le r < 0{,}9$. Следовательно, в следующий момент времени с учетом таблицы переходных вероятностей частица будет в состоянии T_{i+1} = 15.

Допустим в ходе сэмплирования мы получили список из 10 случайных чисел в интервале [0, 1):

[0.467, 0.452, 0.583, 0.604, 0.748, 0.932, 0.609, 0.372, 0.402, 0.026]

Используя эти 10 случайных чисел для формирования выборочных значений наших 10 частиц, получим после полного обновления во времени новый список частиц:

[15,13,13,11,17,15,13,12,12,10].

Обновленный список частиц приводит к соответствующему обновленному распределению степеней уверенности $B(T_{i+1})$:

T_{i+1}	10	11	12	13	14	15	16	17	18	19	20
$B(T_{i+1})$	0.1	0.1	0.2	0.3	0	0.2	0	0.1	0	0	0

Теперь выполним обновление на основе наблюдения, предполагая, что сенсорная модель $Pr(F_i|T_i)$ утверждает, что вероятность правильного прогноза $f_i=t_i$ равна 0.8, а остальные 10 возможных значений состояний предсказываются с вероятностью 0.02. Если наблюдаемый прогноз F_{i+1} = 13, то веса частиц будут следующими:

Частица	1	2	3	4	5	6	7	8	9	10
Состояние	15	13	13	11	17	15	13	12	12	10
Вес	0.02	0.8	0.8	0.02	0.02	0.02	0.8	0.02	0.02	0.02

После суммирования весов каждого из состояний, получим

Состояние	10	11	12	13	15	17
Вес	0.02	0.02	0.04	2.4	0.04	0.02

Суммирование значений всех весов дает сумму 2.54, и мы можем нормализовать таблицу весов, чтобы получить распределение вероятностей, разделив каждую запись на эту сумму:

Состояние	10	11	12	13	15	17
Вес	0.02	0.02	0.04	2.4	0.04	0.02
Нормализованный вес	0.079	0.079	0.0157	0.9449	0.0157	0.079

Последним шагом является повторная выборка (ресэмплирование) из этого распределения вероятностей с использованием того же метода, который мы использовали для выборки во время обновления во времени. Допустим, мы генерируем 10 случайных чисел в диапазоне [0;1) со следующими значениями:

[0.315, 0.829, 0.304, 0.368, 0.459, 0.891, 0.282, 0.980, 0.898, 0.341]

Это дает новый (ресэмплированный) список частиц

[13,13,13,13,13,13,13,15,13,13]

с новым распределением степеней уверенности:

T_{i+1}	10	11	12	13	14	15	16	17	18	19	20
$B(T_{i+1})$	0	0	0	0.9	0	0.1	0	0	0	0	0

Обратите внимание, что сенсорная модель предполагает, что наш прогноз погоды весьма точен и характеризуется высокой вероятностью, равной 0,8. Соответственно, наш новый список частиц согласуется с этим: большинство частиц в результате ресэмплирования получают состояние T_{i+1} = 13.

5.3. Задания для выполнения

В лабораторной работе необходимо создать Пакман-агента, который используют сенсоры для обнаружения невидимых призраков. Такой агент, кроме поиска одиночных призраков сможет охотиться с высокой эффективностью на группы из нескольких движущихся призраков.

Задание 1. Структура сети Байеса

Реализуйте функцию **constructBayesNet** в файле **inference.py**. Функция должна создавать пустую байесовскую сеть со структурой, изображенной на рисунке 5.4. Сеть представляет упрощенный мир игры «Охота на привидения». Узлами сети являются следующие переменные: **Pacman, Ghost0** и **Ghost1** – два приведения, **Obs0** и **Obs1** – оценки расстояний до приведений, набдюдаемые сенсорами Пакмана. Полезно перед написанием кода функции **constructBayesNet** взглянуть на примеры в коде функции **printStarterBayesNet** в файле **bayesNet.py**.

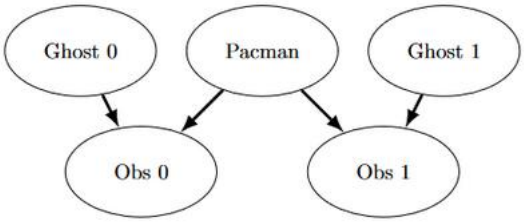


Рисунок 5.4. – Сеть Байеса игры “Охота на привидения”

Чтобы протестировать и отладить свой код, выполните команду:

python autograder.py -q q1

Задание 2. Объединение факторов

Реализуйте функцию **joinFactors** в **factorOperations.py**. Она принимает список входных факторов **factors** и возвращает новый фактор, таблица вероятностей которого вычисляется как произведение соответствующих вероятностей входных факторов. Функция **joinFactors** реализует правила произведения факторов. Например, если у нас есть фактор вида P(X|Y) и другой фактор вида P(Y), то объединение этих факторов даст P(X,Y).

Вот несколько дополнительных примеров того, что может делать функция **joinFactors**:

- **joinFactors** (P(V,W|X,Y,Z),P(X,Y|Z))=P(V,W,X,Y|Z);
- **joinFactors** (P(X|Y,Z),P(Y))=P(X,Y|Z);

- **joinFactors** (P(V|W),P(X|Y),P(Z))=P(V,X,Z|W,Y).

Не следует полагать, что **joinFactors** вызывается для таблиц вероятностей — можно вызвать **joinFactors** и для факторов, строки таблиц которых не дают в сумме 1.

Для тестирования и отладки кода выполните команду:

python autograder.py -q q2

Может быть полезно запустить определенные тесты во время отладки, чтобы увидеть только один набор факторов. Например, чтобы запустить только первый тест, выполните:

python autograder.py -t test_cases/q2/1-product-rule

Задание 3. Исключение переменных (маргинализация)

Реализовать функцию исключения переменных **eliminate** в **factorOperations.py.** Она принимает фактор **factor** и переменную для исключения **eliminationVariable** и возвращает новый фактор, который не содержит эту переменную. Это соответствует суммированию всех строк в таблице вероятностей для **factor**, которые отличаются только значением исключаемой переменной.

Вот несколько примеров того, что может делать функция **eliminate**:

- **eliminate**(P(X,Y|Z),Y)=P(X|Z)
- **eliminate**(P(X,Y|Z),X)=P(Y|Z)

Для тестирования и отладки кода запустите

python autograder.py -q q3

Может быть полезно запустить определенные тесты во время отладки. Например, чтобы запустить только первый тест, выполните:

python autograder.py -t test_cases/q3/1-simple-eliminate

Задание 4. Вывод на основе исключения переменных

Реализуйте функцию вывода для сети Байеса на основе исключения переменных **inferenceByVariableElimination** в **inference.py**. Она формирует ответ на вероятностный запрос, который представляется с помощью списка переменных запроса и свидетельств.

Для тестирования и отладки кода функции выполните команду:

python autograder.py -q q4

Может быть полезно запустить определенные тесты во время отладки. Например, чтобы запустить только первый тест, выполните команду:

python autograder.py -t test_cases/q4/1-disconnected-eliminate

Алгоритм должен перебирать скрытые переменные в порядке исключения, выполняя объединение и исключение переменных, пока не останутся только переменные запроса и свидетельств.

Сумма вероятностей в вашем выходном факторе должна быть равна 1 (чтобы это была истинная условная вероятность, обусловленная свидетельством).

Посмотрите на функцию **inferenceByEnumeration** в **inference.py** для примера того, как использовать дополнительные функции для решения задания. Напомним, что вывод путем перечисления сначала объединяет все переменные, а затем исключает все скрытые переменные. Напротив, вывод на основе исключения переменных чередует объединение и исключение путем итерации по всем скрытым переменным и выполняет объединение и исключение для одной скрытой переменной, прежде чем перейти к следующей скрытой переменной.

Вам нужно будет позаботиться об особом случае, когда фактор, который вы объединили, имеет только одну безусловную переменную (в строках документирования функции указано, что делать в этом случае).

Задание 5а. Класс DiscreteDistribution

К сожалению, наличие в игре Пакман временных шагов приводит к разрастанию графа сети Байеса, чтобы исключение переменных стало жизнеспособным. Вместо этого далее будем использовать прямой алгоритм для точного вывода в CMM и фильтрацию частиц для еще более быстрого, но приближенного вывода.

Для остальной части проекта мы будем использовать класс **DiscreteDistribution**, определенный в **inference.py**, для моделирования распределений. Класс используется для работы с дискретными распределениями. Этот класс является разновидностью словаря Python, где ключами являются значения переменных распределения, а значения ключей равны вероятностям (степени уверенности в возможном значении ключа).

В задании необходимо дописать недостающие методы этого класса: **normalize** и **sample.** Метод **normalize** нормализует значения распределения, таким образом, чтобы сумма всех значений была равна единице. Метод **sample** формирует случайную выборку из распределения в соответствии с алгоритмом, описанным п. 5.2.7.

Задание 5б. Вероятность наблюдения

В этой части задания необходимо реализовать метод **getObservationProb** базового класса **InferenceModule**, определяемого в файле **inference.py**. Метод должен принимать на вход наблюдение (зашумленное значение расстояния до призрака **noisyDistance**), позицию Пакмана **pacmanPosition**, позицию призрака **ghostPosition**, позицию тюремной камеры для призрака **jailPosition** и возвращать вероятность наблюдения **noisyDistance** для заданных положений Пакмана и призрака:

$$P(\text{noisyDistance} \mid \text{pacmanPosition}, \text{ghostPosition}).$$

По сути метод реализует модель наблюдения (восприятия) СММ.

Чтобы протестировать свой код, запустите автооцениватель для этого задания:

```
python autograder.py -q q5
```

Внесите код и результаты тестирования метода в отчет.

Задание 6. Точный вывод на основе наблюдений

В этом задании необходимо реализовать метод **observeUpdate** класса **ExactInference**, определяемого в файле **inference.py**. Метод обновляет распределение степеней уверенности агента в отношении позиций призрака, оцениваемых на основе данных, поступающих от сенсоров Пакмана. Необходимо реализовать онлайн-обновление степеней уверенности в соответствии с (5.19) при получении нового наблюдения **observation** – зашумленного манхеттенского растояния до призрака. Метод **observeUpdate** должен обновлять степени уверенности для каждой возможной позиции призрака после получения наблюдения. Необходимо циклически выполнять обновления для всех значений переменной **self.allPositions**, которая включает в себя все легальные позиции призрака, а также специальную тюремную позицию. Степени уверенности представляются вероятностями того, что призрак находится в определенной позиции, и хранятся в виде объекта **DiscreteDistribution** в поле с именем **self.beliefs**, которое необходимо обновлять.

Задание 7. Точный вывод во времени

В предыдущем задании было реализовано обновление распределения степеней доверия на основе наблюдений. К счастью, наблюдения Пакмана — не единственный источник информации о том, где может быть призрак. Пакман также знает, как может двигаться призрак, а именно, что призрак не может пройти сквозь стену или более чем через одну ячейку за один временной шаг.

Представим следующий сценарий, в котором имеется один призрак. Пакман получает серию наблюдений, которые указывают на то, что призрак очень близко, но затем поступает одно наблюдение, которое указывает, что призрак очень далеко. Наблюдение, указывающие на то, что призрак находится очень далеко, вероятно, является результатом сбоя сенсора. Предварительное знание Пакманом правил движения призрака может снизить влияние этого наблюдения, поскольку Пакман знает, что призрак не может далеко переместиться за один шаг.

В этом задании необходимо реализовать метод **elapseTime** класса **ExactInference**. Метод **elapseTime** должен обновлять степени доверия для каждой возможной новой позиции призрака по истечении одного временного шага в соответствии с (5.18). При этом агент имеет доступ к распределению действий призрака через **self.getPositionDistribution**.

Задание 8. Полное тестирование точного вывода

Теперь, когда Пакман знает, как использовать свои априорные знания о поведении призраков и свои наблюдения, он готов эффективно выслеживать призраков. В задании необходимо будет совместно использовать разработанные методы **observUpdate** и **elapseTime**, а также реализовать простую стратегию жадной охоты. В простой стратегии жадной охоты Пакман предполагает, что призрак находится в наиболее вероятной позиции поля игры в соответствии с его степенью уверенности, и поэтому он движется к ближайшему призраку. До этого момента Пакман выбирал допустимое действие случайно.

Реализуйте метод **ChooseAction** класса **GreedyBustersAgent** в файле **bustersAgents.py**. Ваш агент должен сначала найти наиболее вероятную позицию каждого непойманного призрака, а затем выбрать действие, которое ведет к ближайшему призраку. Чтобы найти расстояние между любыми двумя позициями **pos1** и **pos2**, используйте метод **self.distancer.getDistance(pos1, pos2)**. Чтобы найти следующую позицию после выполнения действия используйте вызов:

successorPosition = Actions.getSuccessor(position, action)

Вам предоставляется список **LivingGhostPositionDistributions**, элементы которого представляют распределения степеней уверенности о позициях каждого из еще непойманных призраков.

При правильной реализации ваш агент должен выиграть игру в тесте **q4/3-gameScoreTest** со счетом выше 700 очков как минимум в 8 из 10 раз.

Задание 9. Инициализация приближенного вывода

В нижеследующих заданиях (10, 11) необходимо реализовать приближенный вероятностный вывод, основанный на алгоритме фильтрации частиц для отслеживания одного призрака.

В данном задании реализуйте методы **initializeUniformly** и **getBeliefDistribution** класса **ParticleFilter** в файле **inference.py**. Частица представляется позицией призрака. В результате применения метода **initializeUniformly** частицы должны быть равномерно (не случайным образом) распределены по допустимым позициям.

Метод **getBeliefDistribution** получает список частиц и отображает позиции частиц в соответствующее распределение вероятностей, представляемое в виде объекта **DiscreteDistribution**. Метод должен возвращать нормализованное распределение.

Задание 10. Приближенный вывод: обновление на основе наблюдения

Необходимо реализовать метод **observUpdate** класса **ParticleFilter** в файле **inference.py**. Метод осуществляет обновление на основе наблюдения в соотвествии с алгоритом, описанным в п. 5.2.10. Наблюдение – это зашумленное манхеттенское расстояние до отслеживаемого призрака. Метод должен выполнять выборку из нормализованного распределения весов частиц и формировать новый список частиц **self.particles**. Вес частицы — это вероятность наблюдения с учетом положения Пакмана и местоположения частицы.

Имеется специальный случай, который необходимо учесть. Когда все частицы получают нулевой вес, список частиц следует повторно инициализировать, вызвав **initializeUniformly**.

Задание 11. Приближенный вывод: обновление во времени

Реализуйте метод **elapseTime** класса **ParticleFilter** в файле **inference.py**. Метод должен сформировать новый список частиц **self.particles** с учетом изменения состояний игры во времени. Используйте алгоритм обновления во времени, описанный в п. 5.2.10.

5.4. Порядок выполнения лабораторной работы

5.4.1. Изучите по лекционному материалу и учебным пособиям [1-3, 9] основные понятия вероятностного вывода, понятие сетей Байеса, марковских моделей, методы и алгоритмы точного и приближенного вероятностного вывода в скрытых марковских моделях. Ответьте на контрольные вопросы.

5.4.2. Используйте для выполнения лабораторной работы файлы из архива **МиСИИ_лаб5_2024.zip.** Разверните программный код лабораторной работы в новой папке и не смешивайте с файлами предыдущих лабораторных работ. Архив содержит следующие файлы:

Файлы для редактирования:	
bustersAgents.py	Агенты для охоты за призраками
inference.py	Код для отслеживания призраков во времени по их шумам.
factorOperations.py	Операции по вычислению новых совместных или маргинальных таблиц вероятностей.
Файлы, которые необходимо просмотреть	
bayesNet.py	Классы BayesNet и Factor
busters.py	Основной файл, из которого запускают игру Ghostbusters с охотниками за призраками (заменяет Pacman.py)
busterGhostAgents.py	Новые агенты-призраки для игры с охотниками за призраками
distanceCalculation.py	Вычисляет расстояния лабиринта
game.py	Вспомогательные классы для Pacman
ghostAgents.py	Агенты, управляющие призраками
graphicsDisplay.py	Графика Pacman
graphicsUtils.py	Графические утилиты
keyboardAgents.py	Интерфейс клавиатуры для управления игрой
layout.py	Код для чтения файлов схем и хранения их содержимого
util.py	Утилиты

Ваш код будет автоматически проверяться автооценивателем. Поэтому не меняйте имена каких-либо функций или классов в коде, иначе вы внесете ошибку в работу автооценивателя.

5.4.3. В рассматриваемом варианте игры цель состоит в том, чтобы выследить невидимых призраков. Пакман оснащен сонаром (слухом), который обеспечивает оценку манхэттенского расстояния до каждого призрака. Игра заканчивается, когда

Пакман выследит и съест всех призраков. Для начала попробуйте сыграть в игру, используя клавиатуру:

python busters.py

Для выхода из игры просто закройте графическое окно.

Цвет позиции поля игры указывает, где может находиться каждый из призраков с учетом оценок расстояний. Оценки расстояний, отображаемые в нижней части графического окна, всегда неотрицательны и всегда находятся в пределах 7 единиц от их истинного значения.

Ваша основная задача — реализовать вероятностный вывод для отслеживания призраков. В случае игры, осуществляемой с помощью клавиатуры, по умолчанию реализуется грубая форма вывода: все квадраты, в которых может быть призрак, закрашиваются цветом призрака. Естественно, нам нужна более точная оценка положения призрака. К счастью, сети Байеса представляют мощный инструмент для максимально эффективного использования имеющейся у нас информации. Вы должны будете реализовать алгоритмы для выполнения как точного, так и приближенного вывода с использованием сетей Байеса.

При отладке кода будет полезно иметь некоторое представление о том, что делает автооцениватель. Автооцениватель использует 2 типа тестов, различающихся файлами **.test**, которые находятся в подкаталогах папки **test_cases**. Для тестов класса **DoubleInferenceAgentTest** вы увидите визуализации распределений, сгенерированных в ходе построения выводов вашим кодом, но все действия Пакмана будут предварительно выбираться в соответствии с ранее заложенной реализацией. Второй тип теста — **GameScoreTest**, в котором действия выбирает созданный вами агент **BustersAgent**. Вы будете наблюдать, как играет Пакман и как он выигрывает.

По мере реализации и отладки кода может оказаться полезным запускать тесты по отдельности. Для этого вам нужно будет использовать флаг **-t** при вызове автооценивателя. Например, если вы хотите запустить только первый тест задания 1, используйте команду:

python autograder.py -t test_cases/q1/1-small-board

Все тестовые примеры можно найти внутри **test_cases/q*.**

Иногда автооцениватель может не сработать при выполнении тестов с графикой. Чтобы определить, эффективен ли ваш код, используйте в этом случае дополнительно при вызове автооценивателя параметр **--no-graphics**, который отключает графику

5.4.4. Просмотрите **bayesNet.py**, чтобы познакомиться с классами, с которыми вы будете работать – **BayesNet** и **Factor**. Вы также можете запустить этот файл, чтобы увидеть пример сети **BayesNet** и связанные с ней факторы:

python bayesNet.py

Вам следует взглянуть на функцию **printStarterBayesNet** – там есть полезные комментарии, которые могут значительно облегчить вам жизнь в дальнейшем. Сеть Байеса, созданная этой функции, является простейшей V-сетью с общим следствием: **(Raining –> Traffic <– Ballgame).**

5.4.5. В задании 1 требуется реализовать функцию **constructBayesNet** в файле **inference.py**. Функция должна создавать сеть, изображенную на рисунке 5.4. Для этого в коде **constructBayesNet** необходимо определить все переменные-узлы сети (**variables**) и ребра (**edges**). Определите каждое ребро сети Байеса, представив его в виде кортежа '**(от, до)**', например: **(GHOST1, OBS1).** Определите возможные значения переменных в словаре **variableDomainsDict[***var***],** для каждой переменной *var*, например: **variableDomainsDict[PAC] = possibleAgentPos,** где **possibleAgentPos** список из кортежей **(x, y)** – возможных позиций Пакмана (**PAC)** в пределах поля игры. Pacman и два призрака могут находиться где угодно (игнорируйте стены на схеме игры). Определите все возможные кортежи позиций призраков с учетом того, что сенсоры Пакмана не точные. Учтите, что оценки наблюдаемых расстояний **OBS** неотрицательны и равны манхэттенским расстояниям от Пакмана до призраков ± шум (**MAX_NOISE**).

5.4.6. В задании 2 требуется реализовать функцию **joinFactors** в **factorOperations.py.** Сформируйте список всех факторов **allFactors** и множества условных **conditional** и безусловных переменных **unconditional:**

```
allFactors=[factor for factor in factors]
conditional=set([])
unconditional=set([])
for factor in allFactors:
    for f in factor.conditionedVariables():
        conditional.add(f)
    for f in factor.unconditionedVariables():
        unconditional.add(f)
```

Сформируйте новый список условных переменных для объединенного результирующего фактора – это те переменные исходного списка **conditional**, которых нет в списке **unconditional**. Создайте новый объединенный фактор, передав на его вход новый список условных переменных

```
newCPT = Factor(unconditional, conditional, variableDomainsDict)
```

Объекты **Factors** хранят **variableDomainsDict**, который сопоставляет каждую переменную со списком значений, которые она может принимать (ее домен). Объект **Factor** получает свой **variableDomainsDict** из сети **BayesNet**. Он содержит все переменные сети **BayesNet**, а не только безусловные и условные переменные, используемые в объектом **Factor**. Для этой задачи вы можете предположить, что все входные факторы взяты из одной и той же сети **BayesNet**, и поэтому все их **variableDomainsDict** одинаковы, т.е.

```
variableDomainsDict=allFactors[0].variableDomainsDict()
```

После создания нового фактора, заполните его таблицу CPT, выполнив вычисления в соответствии с алгоритмом:

Для всех возможных присваиваний assignment нового фактора из newCPT.getAllPossibleAssignmentsDicts():
probability=1
Для каждого входного фактора из allFactors:
вызвать factor.getProbability(assignment) и вычислить вероятности
probability = probability * factor.getProbability(assignment)
newCPT.setProbability(assignment, probability)

5.4.7. В задании 3 требуется реализовать функцию исключения переменных **eliminate(factor: Factor, eliminationVariable: str)** в файле **factorOperations.py.** Функция возвращает новый фактор, из которого удалена переменная **eliminationVariable.**

Помните, что факторы хранят словарь **variableDomainsDict** исходной сети **BayesNet**, а не только безусловные и условные переменные, которые они используют. В результате возвращаемый фактор должен иметь тот же **variableDomainsDict**, что и входной фактор.

Псевдокод решения задания:

1. Формируем новый список безусловных переменных unconditioned без eliminationVariable;
2. Копируем в список условных переменных из фактора factor:
conditioned = factor.conditionedVariables();
3. Извлекаем из variableDomainsDict область значений eliminationVariable:
domain = variableDomainsDict[eliminationVariable];
4. Создаем новый фактор без eliminationVariable:
newFactor = Factor(unconditioned, conditioned, variableDomainsDict);
5. Для каждого возможного присваивания assignment нового фактора, извлекаемого из newFactor.getAllPossibleAssignmentsDicts():
prob = 0;
Для каждого значения elim_val исключаемой переменной из domain:
Копируем assignment в old_assignment;
Расширяем assignment на очередное значение elim_val:
old_assignment[eliminationVariable] = elim_val;
Обращаемся к строке таблицы CPT входного фактора и вычисляем:
prob += factor.getProbability(old_assignment);
Записываем в таблицу CPT нового фактора значение вероятности:
newFactor.setProbability(assignment, prob);
return newFactor

5.4.8. В задании 4 требуется реализовать функцию вывода на основе исключения переменных

inferenceByVariableElimination(bayesNet, queryVariables, evidenceDict, eliminationOrder):

Здесь:

bayesNet – сеть Байеса, на основе которой выполняется вывод;

queryVariables – список безусловных переменных запроса;

evidenceDict – словарь присваиваний **{переменная : значение}** для переменных, которые представлены как свидельства (условные) в запросе вывода;

eliminationOrder – порядок исключения переменных.

Псевдокод решения задания:

```
1. Возвращаем список факторов (с учетом свидетельств) для всех переменных в байесовской сети:
        factors = bayesNet.getAllCPTsWithEvidence(evidenceDict);
2. Для всех переменных var в порядке исключения eliminationOrder:
       #Выполнить объединение факторов по исключаемой переменной var
       factors, new_factor = joinFactorsByVariable(factors, var);
       Если new_factor содержит более одной безусловной переменной:
              Удалить из него исключаемую переменную var;
              Добавить фактор new_factor в конец списка factors;
3. Объединить факторы из factors:  final_factor = joinFactors(factors)
4. Вернуть нормализованный фактор normalize(final_factor)
```

5.4.9. В задании 5а требуется реализовать следующие методы класса **DiscreteDistribution: normalize** и **sample**. Класс является разновидностью словаря языка Python и представляется в виде дискретных ключей и значений пропорциональных вероятностям.

Определите метод **normalize**, который нормализует значения распределения, таким образом, чтобы сумма всех значений была равна единице. Используйте метод **total**, чтобы найти сумму значений распределения. Для распределения, в котором все значения равны нулю, ничего делать не требуется:

```
summa = self.total()
if summa == 0:
   return
```

Иначе необходимо все значения распределения нормализовать по отношению к значению переменной **summa**. Метод должен изменять значения распределения в памяти напрямую, а не возвращать новое распределение.

Определите метод **sample**, который формирует выборку из распределения, в которой вероятность значения ключа пропорциональна соответствующему хранящемуся значению. Предполагается, что распределение не пустое и не все значения равны нулю. Обратите внимание, что обрабатываемое распределение не обязательно нормализовано. Для выполнения этого задания полезной будет встроенная функция Python **random.random()**. Способ формирования выборки из распределения описан в разделе 5.2.5.

Тестов автооценивателя на это задание нет, но правильность реализации можно легко проверить. Для этого можно использовать Python doctests, которые включаются в комментарии определяемых методов. Можно свободно добавлять новые и реализовывать другие собственные тесты. Чтобы запустить **doctest** и выполнить проверку, используйте вызов:

python -m doctest -v inference.py

Обратите внимание, что в зависимости от деталей реализации метода **sample**, некоторые правильные реализации могут не пройти предоставленные doc-тесты. Чтобы полностью проверить правильность метода **sample**, необходимо сделать много выборок и посмотреть, сходится ли частота каждого ключа к соответствующему значению вероятности.

Внесите код и результаты тестирования расзработанных методов в отчет.

5.4.10. В задании 5б необходимо реализовать метод **getObservationProb(self, noisyDistance, pacmanPosition, ghostPosition, jailPosition),** который возвращает вероятность наблюдения **noisyDistance** для заданных позиций Пакмана и призрака. Данный метод соответсвует модели наблюдения (восприятия), которой оснащён Пакман.

Значения, возвращаемые датчиком расстояния, характеризуются распределением вероятностей, которое учитывает истинное расстояние от Пакмана до призрака. Это распределение вычисляется в модуле **busters** функцией **busters.getObservationProbability(noisyDistance, trueDistance)**, которая возвращает вероятности **P(noisyDistance | trueDistance**). Для выполнения задания вы должны использовать эту функцию совместно с функцией **manhattanDistance**, которая вычисляет истинное расстояние **trueDistance** между местоположением Пакмана и местоположением призрака:

```
trueDistance=manhattanDistance(pacmanPosition, ghostPosition)
P=busters.getObservationProbability(noisyDistance, trueDistance)
```

Кроме этого, необходимо учесть особый случай, связанный с арестом призрака. Когда призрак попадает в тюрьму, то датчик расстояния возвращает значение **None**. Если при этом позиция призрака — это позиция тюрьмы, т.е. **ghostPosition == jailPosition**, то датчик расстояния возвращает — **None** с вероятностью **P**=1. И наоборот, если оценка расстояния не **None**, то вероятность нахождения призрака в тюрьме (**ghostPosition == jailPosition**) равна нулю. Соответственно для указанных условий метод **getObservationProb** должен возвращать 1 или 0.

5.4.11. В задании 6 необходимо реализовать метод **observeUpdate** класса **ExactInference**. Метод обеспечивает вычисления в соответствии с правилом обновления (5.19). В данном случае обновляется распределение степеней уверенности агента в отношении позиций призрака, оцениваемых на основе наблюдений, поступающих от датчика расстояний Пакмана. Степени уверенности представляются вероятностями того, что призрак находится в определенной позиции, и хранятся в виде объекта **DiscreteDistribution** в поле с именем **self.beliefs**, которое необходимо обновлять для каждой возможной позиции призрака после получения наблюдения.

Для выполнения задания необходимо использовать функцию **self.getObservationProb** (была определена в задании 5б), которая возвращает вероятность наблюдения с учетом положения Пакмана, потенциального положения призрака и локации тюрьмы. Получить значение позиции Пакмана можно с помощью **gameState.getPacmanPosition()**, позицию тюрьмы с помощью –

self.getJailPosition(), а список возможных позиций призрака с помощью **self.allPositions.** Для выполнения задания вам необходимо реализовать цикл по всем возможным позициям призрака **possibleGhostPos:**

```
for possibleGhostPos in self.allPositions:
    …
```

В цикле должно выполняться обновление степеней уверенности для каждого состояния **self.beliefs[possibleGhostPos]** в соответствии с (5.19) при использовании вероятности наблюдения, вычисляемой в ходе вызова:

```
self.getObservationProb(observation, pacmanPosition,
                                        possibleGhostPos, jailPosition)
```

При этом учтите, что значения $B'(W_{i+1})$ из (5.19) обновляются до значений $B(W_{i+1})$ и все эти значения хранятся в одной и той же области памяти **self.beliefs[possibleGhostPos].** Не забудьте в конце выполнить нормализацию распределения **self.beliefs.normalize().**

Чтобы запустить автооцениватель для этого задания и визуализировать результат используйте команду:

```
python autograder.py -q q6
```

На экране высокие апостериорные степени уверенности представляются более ярким цветом. Если вы хотите запустить этот тест без графики, то используйте вызов с параметром **--no-graphics**:

```
python autograder.py -q q6 --no-graphics
```

Примечание: у агентов-охотников есть отдельный модуль вероятностного вывода для каждого призрака. Вот почему, если печатать наблюдение внутри функции **ObserveUpdate**, то отображается только одно число, даже если имеется несколько призраков.

Внесите код и результаты тестирования метода в отчет.

5.4.12. В задании 7 необходимо реализовать метод **elapseTime** класса **ExactInference**. Метод выполняет вычисления в соответствии с правилом обновления во времени (5.18), где состояния представляются позициями призрака. Чтобы получить распределение степеней уверенности по новым позициям призрака, учитывая его текущую позицию, используйте следующую строку кода:

```
newPosDist = self.getPositionDistribution(gameState, ghostPos),
```

где **ghostPos** — текущая позиция призрака, **newPosDist** — это объект типа **DiscreteDistribution**, в котором для каждой позиции **p** призрака (из **self.allPositions**)

newPosDist[p] — это вероятность того, что призрак будет находиться в момент времени t + 1 в позиции **p**, если в предыдущий момент времени t призрак находился в позиции **ghostPos**.

В ходе реализции метода для каждой позиции призрака **ghostPos** организуйте цикл по всем новым возможным позициям

for newPos, prob in newPosDist.items():

…,

в котором выполните обновление степеней уверенности нахождения призрака в новых позициях **beliefDict[newPos]** в соответствии с (5.18):

beliefDict[newPos]=beliefDict[newPos]+self.beliefs[ghostPos]*prob,

где **beliefDict** – словарь типа **DiscreteDistribution()**. При этом значения $B(w_i)$ соответствуют **self.beliefs[ghostPos],** а переходные вероятности $Pr(W_{i+1},|w_i)$ хранятся в **prob**.

По окончании циклов наобходимо нормализовать распределение **beliefDict** и сохранить его в **self.beliefs=beliefDict**.

Обратите внимание, что этот вызов **self.getPositionDistribution** может быть довольно затратным. Поэтому, если время ожидания исполнения вашего кода истечет, то стоит подумать об уменьшении количества вызовов **self.getPositionDistribution**.

При автооценивании правильности выполнения этого задания иногда используется призрак со случайными перемещениями, а иногда используется **GoSouthGhost.** Последний призрак имеет тенденцию двигаться на юг, поэтому со временем распределение степеней уверенности Пакмана должно начать фокусироваться в нижней части игрового поля. Чтобы увидеть, какой призрак используется для каждого теста, просмотрите файлы **.test.**

Для лучшего понимания задания на рисунке 5.5. изображена сеть Байеса в виде скрытой марковской модели (СММ) для 2-х приведений и 2-х моментов времени. В этом случае **getPositionDistribution** формально соответствует выражению $P(G_{t+1}$ |**gameState**, $G_t)$).

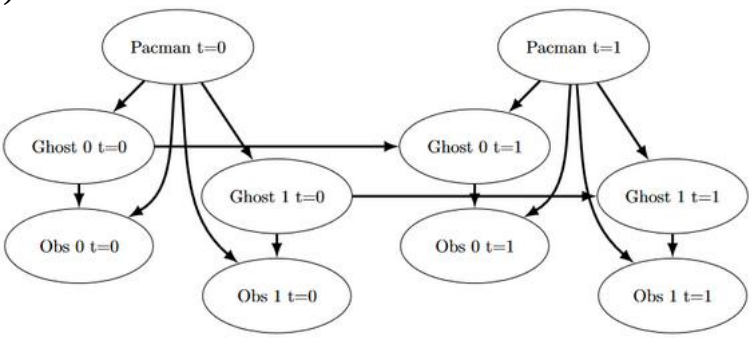


Рисунок 5.5. – Сеть Байеса в виде СММ для 2-х моментов времени

Для автооценивания этого задания и визуализации результатов используйте команду:

python autograder.py -q q7

Если вы хотите запустить этот тест без графики, то добавьте в предыдущий вызов параметр **--no-graphics.**

Наблюдая за результатами автооценивания, помните, что более светлые квадраты указывают на то, что призрак с большей вероятностью будет находиться в них. Подготовьте ответы на вопросы: В каком из тестовых случаев вы заметили различия в закраске квадратов? Можете ли вы объяснить, почему некоторые квадраты становятся светлее, а некоторые темнее?

Внесите код и результаты тестирования метода в отчет.

5.4.13. В начальной части кода задания 8 определяется позиция Пакмана **pacmanPosition**, формируются списки допустимых действий Пакмана **legal** и распределений степеней уверенности о позициях ещё непойманных призраков **livingGhostPositionDistributions**

```
pacmanPosition = gameState.getPacmanPosition()
legal = [a for a in gameState.getLegalPacmanActions()]
livingGhosts = gameState.getLivingGhosts()
livingGhostPositionDistributions =
                [beliefs for i, beliefs in enumerate(self.ghostBeliefs) if livingGhosts[i+1]]
```

Вам следует с помощью списка **livingGhostPositionDistributions** найти наиболее вероятные позиции каждого непойманного призрака и сохранить их, например, в списке **ghostMaxProb**.

Затем в цикле для всех допустимых действий Пакмана с помощью вызова

```
successorPosition = Actions.getSuccessor(pacmanPosition, action)
```

определите следующую позицию после действия **action** и для всех наиболее вероятных позиций призраков из **ghostMaxProb** создайте список **minDist** из пар в виде кортежей (действие, расстояние), где расстояние – это расстояние от Пакмана до призрака:

```
minDist.append((action, self.distancer.getDistance(successorPosition, ghostPos)))
```

Анализируя этот список, найдите минимальное расстояние до призрака

```
minGhostDist = min([d for act, d in minDist])
```

и верните действие **act**, ведущее в сторону ближайшего призрака:

```
for act, d in minDist:
    if d==minGhostDist:
        return act
```

Мы можем представить (рисунок 5.6) как работает наш жадный агент, внеся дполнения в рисунок 5.5.

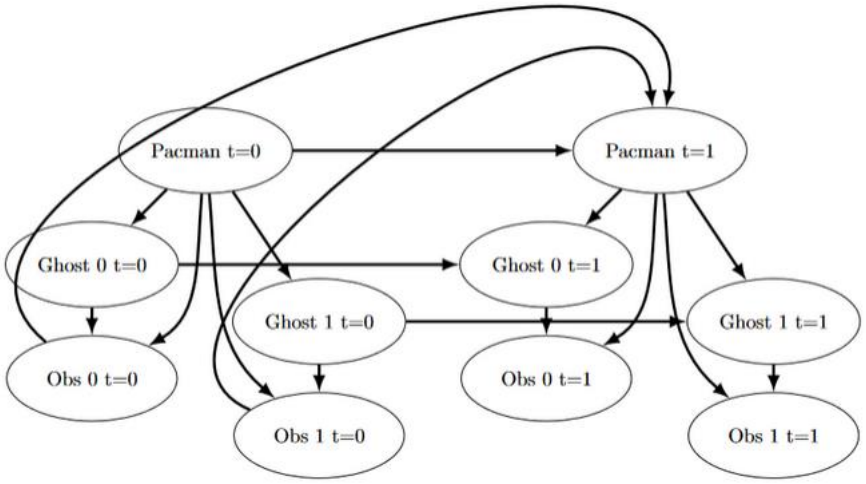


Рисунок 5.6. – Взаимодействия в СММ для 2-х моментов времени

Чтобы запустить автооцениватель для этого задания и визуализировать результат, используйте команду:

python autograder.py -q q8

Если вы хотите запустить этот тест без графики, вы можете добавить параметр **--no-graphics**.

Внесите код метода **ChooseAction** и результаты тестирования в отчет.

5.4.14. При выполнении задания 9 в методе **initializeUniformly** переменная, в которой хранятся частицы, представляется в виде списка **self.particles**, элементами которого являются позиции частиц. Хранение частиц в виде другого типа данных, например, словаря, является неправильным и приведет к ошибкам.

Число инициализирумых частиц задается конструктором класса **ParticleFilter** и по умолчанию равно **self.numParticles**=300. Допустимые позиции частиц **positions** определяются следующим образом: **legal**=**self.legalPositions**.
Псевдоалгоритм решения задачи:

1. **Определить число частиц с помощью particle_num = self.numParticles;**
2. **Определяем допустимые позиции частиц: legal = self.legalPositions;**
3. **Повторять число раз, равное целому числу частиц в одной позиции: (int(particle_num / len(legal))):**
   Расширить список частиц на список legal: self.particles.extend(legal).

Метод **getBeliefDistribution** получает список частиц и формирует соответсвующее распределение. Поэтому в начале создайте переменную-распределение **beliefDist**, которая является экземпляром класса **DiscreteDistribution**. Затем посчитайте число частиц в каждой позиции:

```
for pos in self.particles:
    beliefDist[pos]=beliefDist[pos]+1
```

Нормализуйте полученное распределение **beliefDist** и верните его в качестве результата.

Чтобы протестировать задание выполните команду:

```
python autograder.py -q q9
```

Внесите код разработанных методов и результаты тестирования в отчет.

5.4.15. В задании 10 необходимо сформировать выборку из распределения с учетом весов наблюдений. Поэтому создайте в начале экземпляр распределения, например **weightsDist,** путем вызова конструктора класса **DiscreteDistribution()**.

Чтобы найти вероятности наблюдений с учетом положения Пакмана, потенциального положения призрака и положения тюрьмы, используйте ранее определенную функцию **self.getObservationProb**. В соотвествии с алгоритмом обновления на основе наблюдения (см. п.5.2.10) найдите сумму весов для каждой позиции

```
for pos in self.particles:
   weightsDist[pos]+=
         self.getObservationProb(observation, pacmanPosition, pos, jailPosition)
```

Нормализуйте полученное распределение **weightsDist** и сформируйте новый список частиц, сделав выборки из **weightsDist**:

```
self.particles = [weightsDist.sample() for _ in range(int(self.numParticles))]
```

Не забудьте учесть особый случай, когда все частицы получают нулевой вес. В этом случае следует повторно инициализировать список частиц, вызвав метод **initializeUniformly(gameState)**.

Метод возвращает обновленный список частиц **self.particles.**

Чтобы запустить автооцениватель для этого задания и визуализировать результаты тестирования, выполните команду:

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

или без графики

```
python autograder.py -q q10 --no-graphics
```

Внесите код разработанных методов и результаты тестирования в отчет.

5.4.16. В задании 11 необходимо реализовать в виде метода **elapseTime** класса **ParticleFilter** алгоритм обновления во времени, описанный в п. 5.2.10. Так как в

итоге метод должен формировать новый список частиц как выборку из распределения, то создайте в начале экземпляр распределения, например **elapseDist=DiscreteDistribution()**.

Аналогично методу **elapseTime** класса **ExactInference** для определения следующей возможной позиции частицы по предыдущей позиции **pos** следует использовать функцию **self.getPositionDistribution()**. Тогда распределение новых позиций частиц можно определить так:

```
for pos in self.particles:
   newPosDist = self.getPositionDistribution(gameState, pos)
```

Распределение **newPosDist** представляется словарем с парами занчений { **newPos, prob** }, где **newPos** – новая позиция частицы, а **prob** – вероятность нахождения частицы в этой позиции. Для каждой позиции из списка частиц **self.particles** посчитайте сумму вероятностей нахождения частиц в соответствующей новой позиции и сохраните значения в **elapseDist[newPos]**:

```
for newPos, prob in newPosDist.items():
   elapseDist[newPos]+=prob
```

Нормализуйте полученное распределение **elapseDist** и сформируйте с его помощью новый список частиц

```
self.particles=[elapseDist.sample() for _ in range(int(self.numParticles))]
```

Данный вариант метода **elapseTime** позволяет отслеживать призраков почти так же эффективно, как и в случае точного вывода.

Обратите внимание, что для этого задания автооцениватель тестирует как метод **elapseTime**, так и полную реализацию фильтра частиц, сочетающую **elapseTime** и обработку наблюдений.

Чтобы запустить автооцениватель для этого задания и визуализировать результаты используйте команду:

```
python autograder.py -q q11
```

Для запуска теста без графики добавьте параметр **--no-graphics.** Внесите код разработанных методов и результаты тестирования в отчет.

5.5. Содержание отчета

Цель работы, описание основных понятий сетей Байеса, марковских моделей, скрытых марковских моделей, описание алгоритмов прямого распространения для СММ (правил обновления во времени и на основе наблюдения), описание понятия фильтрации частиц и соотвествующих алгоритмов вывода, код реализованных

функций с комментариями в соответствии с заданиями 1-11, результаты автооценивания заданий, выводы по проведенным экспериментам с разными алгоритмами вероятностных выводов.

5.6. Контрольные вопросы

5.6.1 Объясните, что понимают под степенью уверенности высказываний?

5.6.2. Что понимают под совместным распределением случайных переменных? Свойства совместного распределения?

5.6.3. Что понимают под событием?

5.6.4. Что такое маргинальное распределение? Как его получить из совместного распределения случайных переменных?

5.6.5. Запишите формулу условной вероятности *x* при известном *y*.

5.6.6. Запишите правило произведения для 2-х переменных.

5.6.7. Запишите цепочное правило.

5.6.8. Запишите правило Байеса и объясните его.

5.6.9. Определите понятие независимости 2-х случайных переменных.

5.6.10. Определите понятие условной независимости 2-х случайных переменных при заданной третьей переменной.

5.6.11. Определите понятие сети Байеса.

5.6.12. Запишите выражение полного совместного распределения для сети Байеса и объясните его.

5.6.13. Сформулируйте критерии D-разделенности для различных триплетов подсетей Байеса.

5.6.14. Что понимают под точным и приближенным вероятностным выводом?

5.6.15. Сформулируйте алгоритм формирования случайной выборки из заданного распределения.

5.6.16. Приведите пример марковской модели. Какие предположения независимости используют в марковской модели?

5.6.17. Определите алгоритм прямого распространения для марковской модели.

5.6.18. Определите понятие скрытой марковской модели.

5.6.19. Сформулируйте задачу фильтрации для СММ.

5.6.20. Какие предположения независимости используют в СММ?

5.6.21. Сформулируйте правило обновления во времени и правило на основе наблюдений для СММ.

5.6.22. Сформулируйте алгоритм прямого распростанения для СММ.

5.6.23. Что такое фильтрация частиц применительно к СММ?

5.6.24. Сформулируйте правило обновления во времени и правило на основе наблюдений, применяемые при фильтрации частиц.

6. ЛАБОРАТОРНАЯ РАБОТА № 6 «ИССЛЕДОВАНИЕ НЕЙРОННЫХ СЕТЕЙ»

6.1. Цель работы

Исследование архитектур различных нейронных сетей, функций потерь, методов машинного обучения, основанных на градиентном спуске, приобретение навыков разработки, обучения и программирования архитектур нейронных сетей для решения задач нелинейной регрессии, классификации рукописных цифр и идентификации языка

6.2. Краткие теоретические сведения

6.2.1. Машинное обучение

В машинном обучении системы обучаются, а не программируются явно. В качестве обучаемых систем широко используются модели искусственных нейронных сетей (ИНС), структурированные в виде слоёв. Им передаются многочисленные примеры данных, имеющие отношение к решаемой задаче, а ИНС находят в этих примерах статистическую структуру, которая позволяет вырабатывать правила для автоматического решения задачи. Существуют три вида алгоритмов машинного обучения — это **алгоритмы обучения с учителем** (supervised learning algorithms), **алгоритмы обучения без учителя** (unsupervised learning algorithms), **алгоритмы обучения с подкреплением** (reinforcement learning) [10].

При обучении с учителем предполагается, что для каждого входного вектора ИНС мы заранее знаем желаемый выходной вектор и используем «отличие» действительного выходного вектора от желаемого для изменения в нужном направлении **параметров ИНС** (весов, смещений) , чтобы минимизировать «отличие». При обучении без учителя нам не известна желаемая реакция ИНС на заданный входной вектор. В этом случае ИНС должна «самообучаться», обнаруживая закономерности, присущие пространству входных данных. В этой лабораторной работе рассматриваются алгоритмы обучения с учителем.

В ходе обучения используют **набор данных** (датасет), который разбивают на три подмножества: **обучающее** (training), **валидационное** (validation) и **тестовое** (test) подмножества. Обучающее помножество данных используется для фактического создания модели ИНС, отображающей входные данные в выходные. Валижационное подмножество (данные разработки) используется для оценки эффективности модели ИНС путем вычисления точности предсказаний с помощью ИНС. Если модель ИНС работает не так хорошо, как бы хотелось (для этого вычисляется функция потерь), то можно вернуться и обучить ее снова, скорректировав либо специальные параметры модели, называемые **гиперпараметрами** (скорость обучения, размер мини-батча, коэффициент регуляризации и др.), либо использовать другой алгоритм обучения, пока не будут получены удовлетворительные оценки эффективности ИНС. На заключительном этапе используйте обученную модель ИНС для

предсказания значений тестового подмножества данных и оценки итоговой точности модели. Тестовое подмножество — это часть данных, которая не используется в ходе обучения ИНС.

6.2.2. Линейные классификаторы. Бинарный персептрон

Основная идея линейного классификатора заключается в том, чтобы выполнить классификацию, используя линейную комбинацию признаков $f_i(\mathbf{x})$ классифицируемых данных **x**. В векторной форме мы можем записать это как скалярное произведение вектора весов **w** на вектор признаков данных **f**(**x**):

$$net_w(\mathbf{x}) = \sum_i w_i f_i(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}), \quad (6.1)$$

где $net_w(\mathbf{x})$ – сетевое (предактивационное) значение. Функция формирования вектора признаков **f** соответствует некоторой предварительной обработке данных с целью получения признаковых представлений данных.

Рассмотрим реализацию простого линейного классификатора – бинарного персептрона. **Бинарный персептрон** находит границу решения, которая разделяет обучающие данные на два класса. В этом случае, когда сетевое значение (6.1) для точки данных **x** положительное, персетрон классифицирует эту точку данных как точку с положительной меткой +1, а если значение отрицательное – то как точку с отрицательной меткой -1. В ходе обучения персептрона мы ищем наилучшие возможные веса **w** так, чтобы любая обучающая точка могла быть идеально классифицирована.

Алгоритм обучения бинарного персептрона:

1. Инициализируйте все веса нулевыми значениями: **w** = **0** ;

2. Для каждого обучающей точки данных **x** с признаками **f**(**x**) и истинной меткой класса $y* \in \{-1, +1\}$ повторяйте:

(a) Классифицируйте точку данных **x**, используя текущий вектор весов **w**:

$$y = \begin{cases} +1, если\ net_w(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}) \geq 0 \\ -1, если\ net_w(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}) < 0 \end{cases}, \quad (6.2)$$

где y предсказанная метка;

(b) Сравните предсказанную метку y с истинной меткой y^*:

- если $y = y^*$, ничего не делайте;
- иначе, если $y \neq y^*$, обновите веса: $\mathbf{w} \leftarrow \mathbf{w} + y*\mathbf{f}(\mathbf{x})$.

3. Если для каждой обучающий точки данных все метки предсказаны правильно, то завершите работу. В противном случае повторите шаг 2.

6.2.3. Линейная регрессия

Задачи регрессии — это задача машинного обучения, в которой выход модели y является непрерывной переменной. Признаки могут быть как непрерывными, так и категориальными. Обозначим набор входных признаков модели как **x**

$\in$ R^n т. е. $\mathbf{x} = (x_1, \ldots, x_n)$. Тогда модель прогнозирования, соответствующая линейной регрессии, запишется в виде:

$$net_w(\mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_n x_n, \quad (6.3)$$

где w_i – веса модели, которые необходимо оценить. Вес w_0 называют свобоным членом (в нейронных сетях – смещением) модели. Иногда к вектору признаков $\mathbf{x}$ добавляют признак, значение которого равно 1, чтобы можно было записать модель (6.3) в виде скалярного произведения $\mathbf{w}^T\mathbf{x}$, где $\mathbf{x} \in R^{n+1}$. Для оценки качества обучения модели используется функции потерь L2, которая соответствует среднему квадрату ошибки предсказания [10]

$$Loss(\mathbf{w}) = \frac{1}{2N}\sum_{j=1}^{N}(y^j - net_w(\mathbf{x}^j))^2 = \frac{1}{2N}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2, \quad (6.4)$$

где каждая строка матрицы $\mathbf{X}$ – это вектор $\mathbf{x}^j$, соответствующий j-ой точке данных. Дифференцируя функцию потерь по $\mathbf{w}$ и приравнивая производные нулю, можно найти оценку оптимального вектора весов.

6.2.4. Алгоритмы оптимизации параметров

В общем случае для заданной функции потерь (6.4) может не существовать решения в замкнутой форме. В таких случаях используют градиентные методы для поиска оптимальных весов. Идея заключается в том, что градиент функции указывает направление самого крутого увеличения функции, а направление, обратное градиенту, указывает направление самого крутого спуска. Соответственно, для поиска минимума функции потерь может использоваться **алгоритм градиентного спуска** [10]:

1. Инициализировать веса **w** случайными значениями;
2. **While w** *не сошлось* **do**:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\frac{\partial Loss(\mathbf{w})}{\partial \mathbf{w}}.$$

Здесь α – скорость обучения, $\frac{\partial Loss(\mathbf{w})}{\partial \mathbf{w}}$ – градиент функции потерь. В соответствии с этим алгоритмом обновление весов на каждой итерации выполняют на вектор пропорциональный градиенту функции потерь.

Так как датасет обычно содержит большое количество точек данных, то вычисление градиента на каждой итерации для всех точек данных слишком затратно. Поэтому были предложены такие подходы, как стохастический и мини-блочный градиентный спуск. При **стохастическом градиентном спуске** на каждой итерации алгоритма используется только одна точка данных для вычисления градиента. Эта точка данных каждый раз случайным образом выбирается из набора данных. Учитывая, что мы используем только одну точку данных для оценки градиента, стохастический градиентный спуск может привести к зашумленным градиентам и, таким образом, затруднить сходимость. **Мини-блочный градиентный спуск** является компромиссом между стохастическим и обычным алгоритмом градиентного

спуска, поскольку он использует **мини-блок** (batch) размером *m* точек данных каждый раз для вычисления градиентов. Размер мини-блока *m* является гиперпараметром, выбираемым пользователем.

6.2.5. Логистическая регрессия

Логистическая регрессия позволяет предсказать категориальную переменную. Для этого линейная комбинация входных признаков преобразуется в вероятность с помощью логистической функции:

$$P(y \mid \mathbf{x}; \mathbf{w}) = 1 \Big/ (1 + e^{-\mathbf{w}^T \mathbf{x}}). \quad (6.5)$$

Важно отметить что, хотя логистическая регрессия и называется регрессией, она используется для решения задач классификации, а не задач регрессии. Обратите внимание, что значения функции (6.5) находятся в интервале от 0 до 1. Интуитивно, логистическая функция моделирует вероятность принадлежности точки данных **x** к классу с меткой 1. Например, после того, как мы обучили логистическую регрессию, мы можем вычислить выход логистической функции для новой точки данных. Если значение выхода больше 0,5, мы классифицируем **x** как класс с меткой 1, а в противном случае — с меткой 0.

Оценить веса логистической регрессии можно, используя алгоритм градиентного спуска. При этом в качестве функции потерь используют отрицательную логарифмическую вероятность: – log(*P*(*y*| **x**; **w**)). Интуиция подсказывает, что хорошей (с низкими потерями) является модель, которая назначает высокую вероятность истинному выходу *y*, соответствующему входу **x** [10].

Многоклассовая логистическая регрессия позволяет классифицировать точки данных по *K* различным категориям, а не только по двум. В этом случае мы строим такую модель, которая даёт оценки вероятностей принадлежности точки данных к одной из *K* возможных категорий. Для этого вместо логистической функции используется **softmax** функция, которая определяет вероятность отнесения точки данных с признаками $\mathbf{f}(\mathbf{x}_i)$, к классу *j* как:

$$P(y_i = j \mid \mathbf{f}(\mathbf{x}_i); \mathbf{W}) = \frac{e^{\mathbf{w}_j{}^T \mathbf{f}(\mathbf{x}_i)}}{\sum_{k=1}^{K} e^{\mathbf{w}_k{}^T \mathbf{f}(\mathbf{x}_i)}}, \quad (6.6)$$

где **W** – матрица весов, каждый столбец которой содержит вектор весов $\boldsymbol{w}_k$. Аргументы экспоненты, называемые в данном случае **логитами** (logits), могут быть вычислены с помощью однослойной нейронной сети, содержащей *K* нейронов с весами $\boldsymbol{w}_k$ и вектором входа $\mathbf{f}(\mathbf{x}_i)$. В ходе обучения такой сети необходимо найти такую матрицу **W**, которая минимизирует функцию потерь в виде отрицательного логарифма вероятности

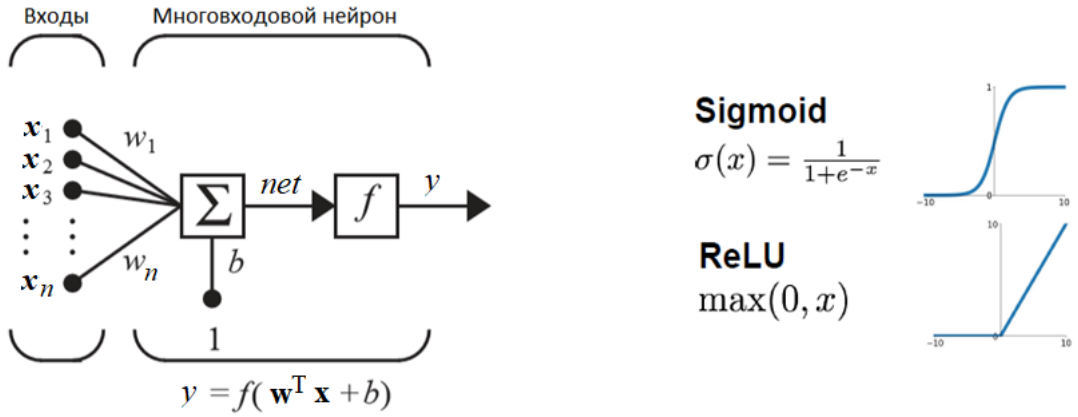
$$Loss(y_i; \mathbf{f}(\mathbf{x}_i)\mathbf{W}) = -\log P(y_i = i \mid \mathbf{f}(\mathbf{x}_i); \mathbf{W}). \quad (6.7)$$

Можно показать, что функция потерь (6.7) соответствует **кросс-энтропии**. Для случая 2-х классов выражение (6.7) соответствует **бинарной кросс-энтропии [**10**]:**

$$H(y_i; p_i) = -[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]. \quad (6.8)$$

6.2.6. Архитектура нейросетей

Элементарной ячейкой нейронной сети является нейрон. Отдельный нейрон с вектором входа **x**, состоящим из n элементов x_1, x_2, …, x_n, изображен на рисунке 6.1а.



а) *б*)

Рисунок 6.1 – Структура нейрона и функции активации

Здесь каждый элемент вектора данных **x** умножается на соответсвующий вес нейрона w_1, w_2, … , w_n Значения элементов вектора **x** , взвешенные с весами **w,** поступают на сумматор. Их сумма равна скалярному произведению вектора $\mathbf{w}^T$ на вектор входа **x**. К этой сумме добавляется смешение b. Результирующая сумма, называемая *сетевым (предактивационным) значением*, равна

$$net = w_1 x_1 + w_2 x_2 + \ldots + w_{1n} x_n + b = \mathbf{w}^T \mathbf{x} + b\,. \quad (6.9)$$

Сетевое значение *net* поступает на вход функции активации f, которая формирует выход нейрона

$$y = f(\mathbf{w}^T \cdot \mathbf{x} + b). \quad (6.10)$$

В качестве функций активации f в нейронных сетях часто используют **логистическую (сигмовидную)** и **линейную выпрямительную (ReLU)** функции, графики которых изображены на рисунке 6.1б.

При изображении нейросетей часто используется обобщенная векторно-матричная схема (рисунок 6.2). Вход нейрона изображается в виде темного прямоугольника, под которым указывается количество элементов n входного вектора **x**. Размер вектора входа **x** указывается ниже символа **x** и равен $n \times 1$. Вектор входа умножается на вектор $\mathbf{w}^T$ длины n. Константа 1 рассматривается как вход, который умножается на скалярное смещение b. Входом функции активации нейрона служит сумма смещения b и произведения $\mathbf{w}^T$ **x**. Эта сумма преобразуется функцией активации f в выходное значение нейрона y, которое в данном случае является скалярной величиной.

Структурная схема, изображенная на рисунке 6.2, соответствует однослойной сети с одним нейроном в слое. Если слой содержит *h* нейронов, то он характеризуется *матрицей весов* связей **W** размером *h* x *n*, вектором смещений **b** и вектором выхода **y** размером *h* x1. В этом случае каждый вход слоя связан с каждым нейроном слоя. Поэтому такой слой называют **полносвязным** (FC - fully-connected).

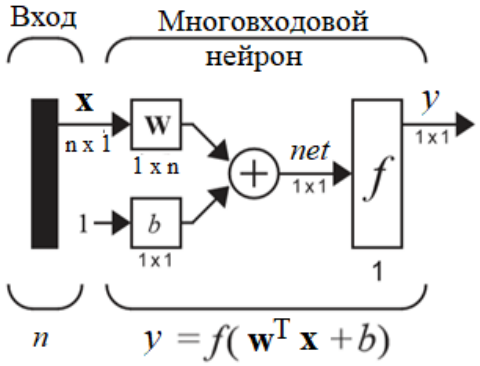


Рисунок 6.2 – Обобщенная векторно-матричная схема нейрона

С помощью схемы однослойной сети можно построить многослойную сеть. В качестве примера на рисунке 6.3 изображена трехслойная сеть прямого распространения (FF – feed forward) с с полносвязными слоями. Для этой сети выход предыдущего слоя является входом следующего слоя. Входом сети является вход первого слоя, т.е. вектор **x**, а выходом – выход последнего слоя, т.е. вектор $\mathbf{y}_3$. Соответственно, прямое распространение входного вектора по сети опишется выражением:

$$\mathbf{y}_3=\mathbf{f}_3(\mathbf{W}_3\mathbf{f}_2(\mathbf{W}_2\mathbf{f}_1(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1)+\mathbf{b}_2)+\mathbf{b}_3. \qquad (6.11)$$

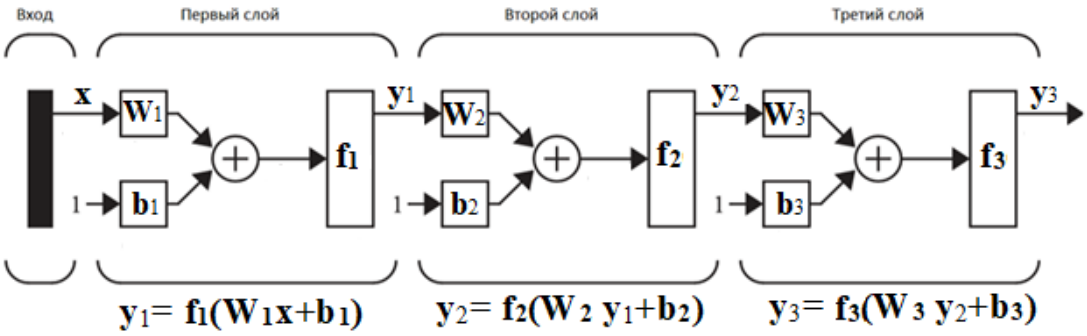


Рисунок 6.3 – Трехслойная сеть прямого распространения

Внутренние слои сети называются **скрытыми слоями** (hidden layers). Вход сети, представленный вектором **x**, называют *входным слоем*. Сокращенно структуру многослойной сети обозначают, указывая последовательно размерность входного слоя и количество нейронов в последующих слоях. Например, структура сети, изображенная на рисунке 6.3, обозначается в виде: n-h_1-h_2-h_3.

Простая нейронная сеть имеет линейные слои, где каждый линейный слой выполняет линейную операцию типа **Wx+b** или $\mathbf{x}^T\mathbf{W}+\mathbf{b}$. Линейные слои разделяются нелинейностями, что позволяет аппроксимировать с помощью нейросети

сложные функции **f(x)**, обеспечивающие автоматическое формирование признаковых представлений входных данных в ходе обучения нейросети. В лабораторной работе будем использовать в качестве нелинейности функцию ReLU, определяемую как relu(x)=max(x,0). Например, простая нейронная сеть с одним скрытым слоем с ReLU нелинейностью и выходным линейным слоем для отображения входного вектора-строки $\mathbf{x}^T$ в выходной вектор признаков $\mathbf{f}(\mathbf{x})$ может быть задана выражением:

$$\mathbf{f}(\mathbf{x})=\text{relu}(\mathbf{x}^T\cdot\mathbf{W}_1+\mathbf{b}_1)\cdot\mathbf{W}_2+\mathbf{b}_2\,, \qquad (6.11)$$

где $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{b}_1$, $\mathbf{b}_2$ – обучаемые параметры нейросети. В этом случае $\mathbf{W}_1$ будет матрицей размера n x h, где n — число элементов входного векторов **x**, а h — размер скрытого слоя (число нейронов скрытого слоя), $\mathbf{b}_1$ — вектор размера h. Мы вольны выбирать любое число нейронов скрытого слоя h (нужно только убедиться, что размеры других матриц и векторов согласованы, чтобы могли выполняться векторно-матричные операции). Использование больших значений h обычно делает сеть более ёмкой (способной вместить больше обучающих данных), но это может затруднить обучение сети (поскольку увеличивает число параметров, которые необходимо обучать), или может привести к переобучению на обучающих данных.

Для повышения вычислительной эффективности градиентного спуска обычно выполняют обработку не одного вектора **x,** а сразу целого блока входных данных. Это означает, что вместо одного входного вектора-строки $\mathbf{x}^T$ размером n на вход нейросети будет подаваться мини-блок (batch) входных данных размером m, представленный в виде матрицы **X** размером $m\times n$.

6.2.6. Фреймворк Pytorch

PyTorch — фреймворк/библиотека для разработки проектов глубокого обучения с упором на гибкость и возможность представлять модели глубокого обучения в характерном для Python стиле [11].

Ниже приведены основные функции, которые вам следует использовать при выполнении лабораторной работы. Этот список не является исчерпывающим. Все функции, которые вы можете использовать, импортированы для вас в модуле **models.py**. Для более детального знакомства с функциями обратитесь к документации PyTorch.

tensor(): Тензоры являются основной структурой данных в Pytorch. Они похожи на многомерные массивы Numpy, в том смысле, что вы можете складывать и умножать их. Каждый раз, когда вы используете функцию Pytorch, вы должны убедиться, что входные данные имеют форму тензора. Вы можете преобразовать список Python в тензор следующим образом: **tensor(data),** где **data** —n-элементный список. Также можно создать тензор на основе массива NumPy; **torch.from_numpy(npy),** где **npy** – массив NumPy.

relu(input): Активационная функция **relu**. Вызывается следующим образом: **relu(input)**. Она принимает входные данные **input** и возвращает **max(input, 0)**.

Linear: Класс для реализации линейного полносвязного нейросетевого слоя. Линейный слой обеспечивает для каждого нейрона слоя вычисление скалярного

произведение вектора весов нейрона и вектора входных данных. Вы должны инициализировать слой в конструктуре нейросети **__init__** следующим образом: **self. lin_layer = Linear(D_in, D_out)**, где **D_in** – число входов слоя, **D_out** – число выходов (нейронов) слоя. Вызов слоя выполняется при запуске модели нейросети с помощью инсрукции: **self. Linear_Layer (input)**. Для линейного слоя, определенного таким образом, Pytorch автоматически создает параметры слоя и обновляет их во время обучения.

movedim(тензор, начальные_позиции_измерений, конечные_позиции_измерений): Функция принимает **тензор** и меняет (переставляет местами) **начальные_позиции_измерений** (переданные как int) на **конечные_позиции_измерений**. Это будет полезно при выполнении задания 3.

cross_entropy(prediction, target): Функция потерь в виде кросс-энтропии (6.7) для многоклассовой классификации (задания 3–5). Здесь **prediction** — предсказанная метка класса моделью нейросети, **target** —истинная метка класса. Чем хуже предсказание нейросети, тем большее значение возвращает функция.

mse_loss(prediction, target): Среднеквадратическая функция потерь (6.4) для задачи регрессии (задание 2). Она используется аналогично функции **cross_entropy**.

Множества данных, используемые в лабораторной работе, доступны в виде объекта **dataset** Pytorch, который необходимо преобразовать в объект **dataloader**, чтобы с его помошью обеспечивать выборку мини-блоков из датасета:

```
data = DataLoader(training_dataset, batch_size = 64)
for batch in data:
   #Здесь размещается код обучения с использованием данных из batch
```

Каждый мини-блок, возвращаемый **DataLoader**, будет представлять собой словарь в форме: **{‘x’:features, ‘label’:label}**, где **label** — это истинная метка (и), которая должна предсказываться нейросетью на основе входных признаков **features**.

6.2.7. Разработка и программирование модели нейросети на PyTorch

Разработка нейронных сетей выполняется методом проб и ошибок. Ниже несколько советов, которые помогут вам при выборе архитектуры нейросети.

1. Будьте систематичны. Ведите журнал каждой архитектуры, которую вы опробовали, какими были гиперпараметры (размеры слоев, скорость обучения и т. д.) и какая эффективность сети была получена. По мере того, как вы набираетесь опыта, вы начнете видеть закономерности того, какие параметры имеют значение.
2. Начните с неглубокой сети (всего один скрытый слой, т. е. одна нелинейность). Более глубокие сети имеют экспоненциально больше комбинаций гиперпараметров, и даже одна ошибка может разрушить процесс обучения. Используйте небольшую сеть, чтобы найти подходящую скорость обучения и размер слоя; после этого вы можете рассмотреть возможность добавления большего количества слоев аналогичного размера.

3. Если скорость обучения выбрана неправильно, то ни один из других вариантов гиперпараметров не будет иметь значения. Слишком низкая скорость обучения приведет к медленному обучению модели, а слишком высокая может привести к тому, что процесс обучения будет расходиться. Начните с проверки разных скоростей обучения, наблюдая, как со временем уменьшаются потери. Для мини-блоков меньщих размеров требуются более низкие скорости обучения. Экспериментируя с размерами мини-блоков, имейте в виду, что наилучшая скорость обучения может для мини-блоков разных размеров отличаться.
4. Воздержитесь от создания слишком широкой сети (слишком большое число нейронов в скрытых слоях). Если вы будете расширять слои, точность будет постепенно снижаться, а время вычислений будет увеличиваться в квадрате. Полный автооценивание для всех частей задания занимает ~12 минут; если ваш код требует намного больше времени, вам следует проверить его на эффективность.
5. Если ваша модель возвращает **Infinity** или **NaN**, то, вероятно, скорость обучения слишком велика.

Рекомендуемые значения для гиперпараметров:

- размеры скрытых слоев: от 100 до 500;
- размер мини-блока: от 1 до 128, для заданий 2 и 3 необходимо, чтобы общий размер набора данных делился нацело на размер мини-блока;
- скорость обучения: от 0,0001 до 0,01;
- количество скрытых слоев: от 1 до 3.

В качестве простейшего примера программирования нейронной сети с помощью PyTorch рассмотрим определение коэффициентов линейной регрессии по набору точек данных. Начнем с четырех точек обучающих данных, полученных с использованием линейной функции $y=7x_0+8x_1+3$:

$$\mathbf{X}=\begin{bmatrix}0 & 0\\ 0 & 1\\ 1 & 0\\ 1 & 1\end{bmatrix},\quad \mathbf{y}=\begin{bmatrix}3\\ 11\\ 10\\ 18\end{bmatrix}.$$

Предположим, что данные предоставлены нам в виде тензоров.

```
>>> x
torch.Tensor([[0,0],[0,1],[1,0],[1,1])
>>> y
torch.Tensor([[3],[11],[10],[18]])
```

Построим и обучим модель вида $f(\mathbf{x})=x_0\cdot w_0+x_1\cdot w_1+b$. Если всё будет сделано правильно, мы должны будем получить, что $w_0=7$, $w_1=8$ и $b=3$.

Сначала создаем массивы обучаемых параметров модели нейросети. В матричной форме они могут быть записаны следующим образом:

$$\mathbf{W}=[\,w_0\ w_1],\quad \mathbf{b}=[b].$$

Для их хранения создаем тензоры в PyTorch:

```
w = Tensor(2, 1)
b = Tensor(1, 1)
```

Тензоры инициализируются нулевыми значениями. Поэтому если мы выведем значения **w** и **b**, то получим:

```
>>> w
torch.Tensor([[0],[0]])
>>> b
torch.Tensor([[0]])
```

Далее вычисляем предсказания *y* для нашей модели. Для этого сначала определяем линейный слой в конструкторе нейросети **__init__()**, как было указано выше с использованием класса **Linear**. Затем вычисляем предсказание

```
predicted_y = self.lin_layer(x)
```

Наша цель — добиться соответствия предсказанных значений **predicted_y** предоставленным данным **y.** В линейной регрессии мы делаем это путем минимизации средненго квадрата потерь, определяемых выражением (6.4). На PyTorch вычисление функции потерь (6.4) запишется следующим образом:

```
loss = mse_loss(predicted_y, y)
```

Для обучения нейронной сети нужно сначала инициализировать оптимизатор. В Pytorch встроено несколько оптимизаторов. Для этой лабораторной работы используйте **алгоритм оптимизации Adam** (вариант градиентного спуска с адаптацией скорости обучения):

```
optim.Adam(self.parameters(), lr=lr),
```

где **lr** — скорость обучения, **self.parameters()** — параметры нейросети, определяемые PyTorch автоматически, после создания экземпляра класса нейросети с помощью его конструктора.

Далее в цикле для каждой итерации по данным необходимо повторять следующие действия:

- обнулить градиенты, рассчитанные PyTorch с помощью вызова **optimizer.zero_grad()** ;
- вычислить тензор потерь **loss**, вызвав функцию **get_loss(),** которая вычисляет предсказание **predicted_y** и вызывает **mse_loss;**
- вычислить градиенты, используя вызов **loss.backward()**;
- обновить парамеры нейросети, вызвав **optimizer.step()**.

6.3. Задания для выполнения

Задание 1. Персептрон

В этом задании необходимо реализовать бинарный персептрон. Ваша задача — завершить реализацию класса **PerceptronModel** в **models.py**.

Для персептрона выходные метки должны иметь значение либо +1, либо −1.

Задание 2. Нелинейная регрессия

В этом задании необходимо разработать и обучить нейронную сеть аппроксимации функцию sin(x) на интервале $[-2\pi,2\pi]$. Вам нужно будет завершить реализацию класса **RegressionModel** в **models.py**.

Для этой задачи достаточно относительно простой архитектуры нейросети (см. рекоменлации в п. 6.2.7). В качестве функции потерь используйте функцию **mse_loss**.

Задание 3. Классификация цифр

В этом задании необходимо обучить сеть классификации рукописных цифр из набора данных MNIST. Каждая цифра представляется изображением размером 28 на 28 пикселей, значения которых хранятся в 784-мерном векторе в виде чисел с плавающей точкой. Выход нейросети представляет собой 10-мерный вектор, который имеет нули во всех позициях, за исключением единицы в позиции, соответствующей правильному классу цифры.

Завершите реализацию класса **DigitClassificationModel** в файле **models.py**. Значение, возвращаемое методом **DigitClassificationModel.run()** должно быть тензором размером **batch_size x 10**, содержащим оценки принадлежности цифры к определенному классу (0-9), где более высокие оценки указывают на более высокую вероятность принадлежности. В качестве функции потерь Вам следует использовать **cross_entropy**. Не включайте ReLU в последний слой сети, который должен быть линейным слоем.

Задание 4. Определение языка

Определение языка — это задача определения языка, на котором написан некоторый текст, по фрагменту текста. Например, ваш браузер может определить язык посещенной веб-страницы и предложить её перевести.

В этом задании необходимо построить модель нейронной сети, которая определяет язык очередного слова, поступающего на вход. Набор данных состоит из слов на пяти языках, как указано в таблице ниже:

Слово	**Язык**
discussed	Английский
eternidad	Испанский
itseänne	Финский
paleis	Голландский
mieszkać	Польский

Разные слова состоят из разного количества букв, поэтому модель нейросети должна иметь архитектуру, которая может обрабатывать слова переменной длины.

В этом задании на вход нейросети будет подаваться отдельные символы слова: x_0, x_1,⋯ , x_{L-1}, где L — длина слова.

Задание 5. Сверточная сеть

В этом задании создайте новую нейросеть для распознавания рукописных цифр в виде класса **DigitConvolutionalModel()** в **models.py**, сначала реализовав функцию **Convolve**. Эта функция принимает входную матрицу данных и матрицу весов и вычисляет их свертку.

6.4. Порядок выполнения лабораторной работы

6.4.1. Изучите по лекционному материалу и учебным пособиям [1-3, 9] основные понятия вероятностного вывода, понятие сетей Байеса, марковских моделей, методы и алгоритмы точного и приближенного вероятностного вывода в скрытых марковских моделях. Ответьте на контрольные вопросы.

6.4.2. Используйте для выполнения лабораторной работы файлы из архива **МиСИИ_лаб6_2024.zip.** Разверните программный код лабораторной работы в новой папке и не смешивайте с файлами предыдущих лабораторных работ. Архив содержит следующие файлы:

Файлы для редактирования:	
`models.py`	Модели персептрона и нейронных сетей для различных приложений.
`nn.py`	Мини-библиотека нейронных сетей.
Файлы, которые необходимо просмотреть	
`nn.py`	Мини-библиотека нейронных сетей.
`autograder.py`	Автооцениватель
`backend.py`	Бэкэнд-код для различных задач машинного обучения.
`data`	Наборы данных для классификации цифр и идентификации языка.

Ваш код будет автоматически проверяться автооценивателем. Поэтому не меняйте имена каких-либо функций или классов в коде, иначе вы внесете ошибку в работу автооценивателя.

6.4.3. Перед началом выполнения задания убедитесь, что у вас установлены библиотеки numpy, matplotlib и PyToch! Если Вы используете дистрибутив **conda**, то можно проверить список пакетов, установленных в текущей среде, набрав команду

conda list

Убедитесь, что в списке уставноленных пакетов имеются указанные выше пакеты.

6.4.4. Для решения задания 1 вам необходимо выполнить рекомендации, указанные ниже.

Дополните конструктор **init(self, dimensions)**. Он должен инициализировать веса персептрона. Убедитесь, что веса сохраняются в виде объекта **Parameter()** с

размерностью **1 x dimensions**. Это необходимо для того, чтобы автооцениватель, а также PyTorch, могли распознать вектор весов как параметр модели нейросети.

Реализуйте метод **run(self, x)**. Он должен вычислять скалярное произведение вектора весов и вектора входа, возвращая объект типа Tensor. Для вычисления скалярного произведения переменных, представленных тензорами, воспользуйтесь функцией PyTorch **tensordot(x,w,dims)**, где **dims** – кортеж из списков, указывающий индексы измерений вдоль которых вычиляется скалярное произведение.

Реализуйте метод **get_prediction(self, x)**, который должен возвращать 1, если скалярное произведение положительное и -1 – если отрицательное.

Запрограммируйте метод **train(self)**. Он должен многократно проходить по набору данных и обновлять параметры на примерах данных, которые были неправильно классифицированы. Когда проход по всему набору данных завершается без ошибок и достигается 100% точность обучения, обучение можно прекратить. Используйте алгоритм обучения бинарного персептрона, приведенный в п. 6.2.2.

Pytorch упрощает выполнение операций с тензорами. Обновление вектора весов пропорционально некоторому направлению, определяемому тензором **direction**, можно выполнить следующим образом:

self.w += direction * magnetic

Для этого задания, как и для всех остальных, каждый мини-блок данных, возвращаемый **DataLoader**, будет представляться в виде словаря: **{'x':features, 'label':label}**, где **label** — это метка, которая должна быть предсказана на основе признаков. Порядок работы с **DataLoader** изложен в п. 6.2.7.

Чтобы протестировать реализацию, выполните автооценивание:

python autograder.py -q q1

Автооценивателю необходимо не более 20 секунд для проверки решения. Если автооцениватель требует значительно большего времни то, вероятно, ваш код содержит ошибку.

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.5. В задании 2 вам необходимо реализовать методы, указанные ниже.

Реализуйте конструктор **RegressionModel.__init__** с необходимой инициализацией слоев нейросети. Используйте сеть не более, чем с двумя скрытыми слоями. Число нейронов в скрытом слое должно быть достаточно большим (100 – 200). Выходной слой должен быть линейным.

Реализуйте метод **RegressionModel.forward** который возвращает тензор размером **batch_size x 1** с предсказанными значениями функции sin(x). Метод просто выхывает слои описанные в конструкторе **RegressionModel.__init__.**

Реализуйте метод **RegressionModel.get_loss**, возвращаюший средний кварат для ошибки предсказанных значений функции sin(x). Этот метод должен вычислять предсказние **y_pred** с помощью **RegressionModel.forward** и вызывать функцию **mse_loss(y_pred, y)** для вычисления и возврата значений потерь.

Реализуйте метод **RegressionModel.train**, который выполнит обучение модели нейросети с использованием обновлений на основе градиента. Организация цикла обучения с использованием возможностей PyTorch изложена в п. 6.2.7.

Набор данных для этого задания содержит только обучающие данные. Для остановки обучения в качестве критерия используйте значение функции потерь. Ваша реализация задания должна обеспечить значение среднего квадрата ошибки обучения **train_loss** не выше 0.02, вычисляемого путем усреднения по всему набору обучающих данных. Для вычисления **train_loss** адаптируйте код:

```
data_x = torch.tensor(dataset.x,dtype=torch.float32)
labels = torch.tensor(dataset.y, dtype=torch.float32)
train_loss = model.get_loss(data_x, labels)
train_loss = train_loss.item()
```

Чтобы протестировать реализацию, выполните автооценивание:

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

Обучение модели должно занять несколько минут.

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.6. В задании 3 необходимо завершить определение класса **DigitClassificationModel.** В целом структура кода этого класса аналогично заданию 2. Отличие состоит в том, что для задания 3, а также задания 4, в дополнение к обучающим данным, имеются валидационное и тестовое множества даных. Вам необходимо использовать **dataset.get_validation_accuracy()** для вычисления точности модели на вадидационных данных. Вычисленная точность необходима для принятия решения о прекращении обучения. Не используйте тестовый набор данных, он используется только во время автооценивания.

Ваша модель должна достичь точности не менее 97% на тестовом наборе данных. Обратите внимание, что автооцениватель оценивает точность модели на основе тестовых данных, в то время как у вас имеется доступ только к точности на валидационном множестве данных. Поэтому, если валидационная точность соответствует порогу 97%, вы все равно можете не пройти автооценивание. Рекомендуется установить немного более высокий порог остановки для валидационной точности, например, 97,5%.

Чтобы протестировать вашу реализацию, запустите автооцениватель:

```
python autograder.py -q q3
```

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.7. В задании 4 необходимо построить модель нейронной сети, которая определяет язык слова в наборе данных. На вход нейросети подаются отдельные символы слова: $\mathbf{x}_0$, $\mathbf{x}_1$,⋯ , $\mathbf{x}_{L-1}$, где L — длина слова. При этом каждый символ представляется “one-hot” бинарным вектором, содержащим 1 в позиции, соответсвующей номеру буквы в общем алфавите распознаваемых языков.

Начнем с применения начальной сети f_{initial}, которая похожа на сети из предыдущих заданий. Она принимает на свой вход символ $\mathbf{x}_0$ и вычисляет некоторый выходной вектор скрытого состояния $\mathbf{h}_1$ размерностью d:

$$\mathbf{h}_1 = f_{\text{initial}}(\mathbf{x}_0)$$

Далее мы объединим выход предыдущего шага со следующей буквой в слове, сгенерировав векторное представление первых двух букв слова. Для этого мы применим подсеть, которая принимает на вход новую букву и формирует новое скрытое состояние с учетом предыдущего скрытого состояния $\mathbf{h}_1$. Обозначим эту подсеть как f:

$$\mathbf{h}_2 = f(\mathbf{h}_1, \mathbf{x}_1)$$

Эти действия повторяем для всех букв входного слова. При этом скрытое состояние на каждом новом шаге аккумулирует представления всех букв, обработанных сетью на данный момент:

$$\mathbf{h}_i = f(\mathbf{h}_{i-1}, \mathbf{x}_{i-1}) \text{ и т.д.}$$

Во всех этих вычислениях функция $f(\cdot,\cdot)$ соответствует одной и той же нейронной сети и использует те же обучаемые параметры, которые определены в f_{initial} . Таким образом, все параметры, используемые при обработке слов разной длины, являются общими.

Вы можете реализовать указанный процесс с помощью цикла **for**, перебирая элементы предоставляемого списка **xs** с кодами символов. На каждой итерации цикла вычисляет либо f_{initial}, либо f. Описанная выше сеть называется **рекуррентной нейронной сетью** (RNN) и изображена на рис.6.4 [11]. Здесь RNN используется для отображения слова «cat» в вектор скрытого состояния $\mathbf{h}_3$ фиксированной длины.

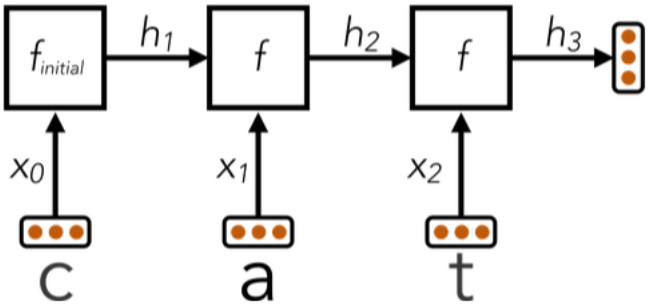


Рисунок 6.4 – Рекуррентная нейронная сеть

После того, как RNN обработает всю входную последовательность, она закодирует слово произвольной длины в виде вектора $\mathbf{h}_L$ фиксированного размера, где L — длина слова. Этот векторный эквивалент входного слова теперь можно обработать дополнительными выходными слоями для классификации языка слова.

Хотя приведенные выше выражения относятся к одному слову, на практике для эффективности необходимо использовать мини-блоки слов. Для упрощения решения задания, предоставляемый код гарантирует, что все слова в одном мини-блоке имеют одинаковую длину. В этом случае скрытое состояние $\mathbf{h}_i$ представляется матрицей $\mathbf{H}_i$ размерности **batch_size** x **d**.

При проектировании RNN учтите следующие рекомендации.

Начните с произвольной архитектуры $f_{\text{initial}}(x)$, аналогичной предыдущим заданиям, при условии, что она имеет хотя бы одну нелинейность.

Рекомендуется использовать следующий подход построения сети $f(\cdot,\cdot)$. Первый слой начальной подсети f_{initial} должен обеспечивать умножение вектора $\mathbf{x}_0$ на матрицу весов $\mathbf{W}_x$ для получения $\mathbf{h}_1 = f(\mathbf{x}_0 \cdot \mathbf{W}_x)$ Для обработки последующих букв следует заменить это вычисление на $\mathbf{h}_i = f(\mathbf{x}_i \cdot \mathbf{W}_x + \mathbf{h}_{i-1} \cdot \mathbf{W}_h)$. Рекомендуется в качестве нелинейности f использовать функцию ReLu. В этом случае вам следует заменить вычисление первого шага **h=relu(self.lin_layer1(xs[0]))** на вычисление вида **h=relu(self.lin_layer1(x) + self.lin_layer2(h))**, где слои **lin_layer1** и **lin_layer2** реализуют умножение векторов **x** и **h** на матрицы весов $\mathbf{W}_x$ и $\mathbf{W}_h$, соответственно.

Скрытый размер d должен быть достаточно большим. Цикл обучения реализуется аналогично заданиям 2 и 3. В качестве функции потерь используйте **cross_entropy(y_pred,y).**

Начните с неглубокой сети f и определите подходящие значения d для вектора скрытого состояния и скорости обучения, прежде чем делать сеть глубже. Если вы сразу начнете с глубокой сети, у вас будет экспоненциально больше комбинаций гиперпараметров, и любой неправильный гиперпараметр может привести к резкому снижению эффективности сети.

Для решения задания вам необходимо завершить реализацию класса **LanguageIDModel.** После обучения нейросеть должна достичь точности не ниже 81% на тестовом наборе данных.

Набор данных задания был создан с помощью автоматизированной обработки текста. Он может содержать ошибки. Тем не менее, эталонная реализация нейросети может правильно классифицировать более 89% слов тестового набора.

Чтобы протестировать вашу реализацию, запустите автооцениватель:

python autograder.py -q q4

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.8. В задании 5 необходимо разработать и обучить сверточную нейронную сеть. Рассмотрим основы построения сверточных слоев.

Часто при обучении нейронной сети возникает необходимость использовать более сложные слои, чем простые линейные слои. Одним из распространенных типов слоев является сверточный слой. Сверточные слои позволяют лучше учесть пространственную информацию, содержащуюся в многомерных входных данных. Например, рассмотрим следующие входные данные, представляющие изображение в виде двумерной матрицы:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{d1} & x_{d2} & \dots & x_{dn} \end{bmatrix}.$$

Если бы мы использовали линейный слой, подобный тому, который использовался в задании 2, то для подачи этих данных на вход нейронной сети нам пришлось бы преобразовать их в следующую форму:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} & \dots & x_{dn} \end{bmatrix}.$$

Но в задачах классификация изображений лучше сохранить пространственную информацию, содержащуюся в исходных данных, и обрабатывать их в матричной форме. Для этого применяют **сверточные слои (**convolutional layers). Двумерный сверточный слой хранит веса нейронов в виде двумерной матрицы, например:

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}.$$

При подаче входных данных слой выполняет свертку матрицы входа и матрицы весов. После выполнения этого **сверточная нейронная сеть** (CNN – Convolutional Neural Network) может преобразовать выход сверточного слоя к одномерному вектору и передать его полносвязным слоям для дальнейшей обработки.

Выходные данные двумерной свертки можно следующим образом:

$$\mathbf{Y} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ y_{d1} & y_{d2} & \cdots & y_{dn} \end{bmatrix},$$

где любой элемент y_{ij} получается в результате поэлементного умножения матрицы **W** и части входной матрицы **X**, которая начинается с элемента x_{ij} и имеет ту же ширину и высоту, что и матрица **W**. Затем все элементы промежуточной матрицы произведений суммируются для вычисления y_{ij}. Например, для нахождения значения y_{22} необходимо поэлементно умножить **W** на следующую часть матрицы входа:

$$\begin{bmatrix} x_{22} & x_{23} \\ x_{32} & x_{33} \end{bmatrix}.$$

В реультате получим матрицу произведений

$$\begin{bmatrix} x_{22} * w_{11} & x_{23} * w_{12} \\ x_{32} * w_{21} & x_{33} * w_{22} \end{bmatrix}.$$

Выполнив суммирование всех элементов этой матрицы, вычислим y_{22}:

$$y_{22} = x_{22} * w_{11} + x_{23} * w_{12} + x_{32} * w_{21} + x_{33} * w_{22}$$

Иногда при применении свертки входная матрица дополняется по краям нулями, чтобы размер выходной матрицы сверточного слоя совпадал с размером входной матрицы. В этом задании вам это не требуется делать. Поэтому ваша выходная матрица должна быть меньше входной матрицы. В общем случае для вычисления размеров выходной матрицы используются варажения:

Ht = 1 + (H + 2 * pad - HH) / stride
Wt = 1 + (W + 2 * pad - WW) / stride

Здесь **H, W** – высота и ширина матрицы входных данных; **Ht, Wt** – высота и ширина матрицы выходных данных; **HH, WW** – высота и ширина окна (матрицы весов) свертки; **stride** – шаг смещения окна свертки (обычно 1) при его наложении на мат-

рицу входных данных; **pad** – число строк или столбцов добаленных нулей к исходной матрице данных (в лабораторной работе 0). Обратите внимание, что размеры выходной матрицы данных должны быть целыми значениями.

В этом задании вам необходимо сначала дописать функцию **Convolve** в **models.py**. Эта функция принимает входную матрицу и матрицу весов и вычисляет их свертку. Обратите внимание, что входная матрица всегда больше матрицы весов и всегда будет передаваться по одной за раз, поэтому вам не нужно выполнять свертку нескольких входных матриц одновременно. При вычислении каждого элемента y_{ij} выходной матрицы вы должны накладывать левый верхний угол окна свертки на соответствующую позицию входной матрицы, вырезать из нее подматрицу **x_slice**

```
x_slice = input[i:(i + HH), j:(j + WW)]
```

и проводить вычисление свертки путем вычисления суммы поэлентных произведений **x_slice** и матрицы весов **weight** (см. выше):

```
y[i, j] = tensordot(x_slice,weight,dims=2)
```

После определения функции **Convolve** завершите определение класса **DigitConvolutionalModel()** в **models.py**. Можно повторно использовать большую часть кода задания 3.

Автооцениватель сначала проверит функцию **Convolve**, чтобы убедиться, что она правильно вычисляет свертку двух матриц. Затем он протестирует вашу модель, чтобы увидеть, может ли она достичь точности 80% на упрощенном подмножестве данных MNIST. Поскольку это задание касается проверки функции **Convolve()**, которую вы написали, то модель должна обучаться относительно быстро.

Сверточная сеть, разработанная в этом задании, скорее всего, будет работать медленнее по сравнению со сверточными слоями, предоставляемыми PyTorch. Это ожидаемо, так как пакеты вроде Pytorch используют оптимизацию для ускорения вычисления сверток.

В методе **run()** мы уже реализовали вызов сверточного слоя и преобразование его матричного выхода в вектор. Далее вам необходимо обработать этот вектор линейными слоями, аналогично заданию 3. Вам понадобится всего пара слоев, чтобы достичь точности 80%.

Чтобы протестировать вашу реализацию, запустите автооцениватель:

```
python autograder.py -q q5
```

Внесите код разработанных методов и результаты тестирования в отчет.

6.5. Содержание отчета

Цель работы, структурные схемы исследуемых нейросетей в соответствии с заданиями 1-5, описание используемых данных и функций потерь, код реализован-

ных функций м методов с комментариями в соответствии с заданиями 1-5, результаты автооценивания заданий, выводы по проведенным экспериментам с разными нейронными сетями.

6.6. Контрольные вопросы

6.6.1 Назовите виды алгоритмов обучения? Объясните, что понимают под обучением с учителем? Что понимают под обучением без учителя?

6.6.2. На какие подмножества разбивают наборы данных, используемые при разработке нейросетей? Назовите назначение каждого из подмножеств?

6.6.3. Что такое гиперпарамеры алгоритма обучения?

6.6.4. Нарисуйте схему бинарного персептрона и объясните алгоритм его обучения? Что представляет собой граница решения бинарного персептрона и как её построить?

6.6.5. Запишите модель прогнозирования линейной регрессии. Как оценивается качество модели линейной регрессии?

6.6.6. Объясните алгоритм градиентного спуска? Сформулируйте отличия стохастического градентного спуска и мини-блочного градиентного спуска?

6.6.7. Запишите выражения логистической регрессии? Запишите выражения многоклассовой логистической регрессии? Что такое softmax? Что называют логитами? Запишите выражение функции потерь, которую называют кросс-энтропией? Запишите выражение бинарной кросс-энтропии?

6.6.8. Нарисуйте схему формального нейрона и запишите выражение для его выхода? Какие активионные функции вам известны? Постройте их графики и запишите выражения?

6.6.9. Нарисуйте схему многослойной полносязной сети прямого распространения и запишите уравнения прямого распространения данных? Что называют скрытым слоем? Как обозначают структуру нейронной сети с помощью символов?

6.6.10. Охарактеризуйте назначение пакета Pytorch? Что такое тензор в PyTorch? Как его определить? Объясните назначение, определение и использование класса **Linear**? Объясните функции **movedim(), cross_entropy(), mse_loss()**? Объясните как с помощью объекта **DataLoader** обеспечить загрузку данных и выборку мини-блоков?

6.6.11. Напишите на Pytoch типовой цикл обучения простейшей 2-х слойной нейросети? Как в Pyrtoch осуществляется вычисление градиентов и применение их обновления параметров нейросети?

6.6.12. Нарисуйте структуру простейшей рекуррентной сети и объясните выражения, лежащие в основе ее функционирования?

6.6.13. Нарисуйте и объясните архитектуру RNN для распознавния языка?

6.6.14. Обясните принцип построения сверточного слоя и алгоритм вычисления свертки?

7. ЛАБОРАТОРНАЯ РАБОТА № 7 «ИССЛЕДОВАНИЕ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ»

7.1. Цель работы

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

7.2. Краткие теоретические сведения

7.2.1. Недетерминированный поиск

Рассмотрим методы поиска в условиях, когда действие агента, выполненное в некотором состоянии, может приводить к нескольким новым состояним с определенной долей вероятности. Такие задачи поиска, в которых поведение агента характеризуется долей неопределенности, относятся к **недетерминированным задачам поиска**. Они могут быть решены с помощью моделей, известных **как марковские процессы принятия решений (Markov Decision Processes – MDP).**

7.2.2. Марковские процессы принятия решений

Марковский процесс принятия решений определяется следующим набором множеств и функций [3, 7, 8]:

1**. Множество состояний** $s \in S$. Состояния в MDP представляются также, как и состояния при поиске решений задач в пространстве состояний;

2. **Множество действий** $a \in A$. Действия в MDP также представляются аналогично действиям в ранее рассмотренных методах поиска решений задач;

3. **Функция перехода** $T(s,a,s')$ **(transition function),** представляется вероятностями перехода агента из состояния s в состояние s' посредством выполнения действия a, т.е. $P(s'|\, s, a)$;

4. **Функция вознаграждения** $R(s,a,s')$ **(reward function)** - определяет *награду*, получаемую агентом при переходе из состояния s в состояние s' посредством выполнения действия a;

5. **Коэффициент дисконтирования** γ **(discount factor) –** фактор определяющий степень важности текущих и будущих наград.

Формально марковский процесс принятия решений можно представить в виде множества

$$\mathbf{MDP=\{}\, S, A, T, R, \gamma \,\mathbf{\}.}$$

При определении MDP также указывают н**ачальное состояние** и, возможно, **конечное состояние**. Награда может быть положительной или отрицательной в зависимости от того, приносят ли действия пользу агенту.

Для MDP характерно выполнение **марковского свойства**: следующее состояние определяется только текущим состоянием. Это подобно поиску, в котором

функция-приемник использует только предшествующее состояние (а не историю состояний).

Решение задачи, представляемой в виде MDP, сводится к построению функции **оптимальной политики:**

$$\pi^*: S \rightarrow A.$$

Политика – это функция **π**, определяющая для каждого состояния *s* действие *a* . Оптимальная политика, обозначаемая как π*, максимизирует ожидаемое накопленное вознаграждение, если ей следовать. В заданном состоянии *s* агент, следующий политике π, выберет действие $a=\pi(s)$.

Смену состояний агента можно представить в виде последовательности:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

Цель функционирования агента – максимизация суммарного вознаграждения, которое можно представить в виде **функции полезности**

$$U([s_0, a_0, s_1, a_1, s_2, \dots]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + R(s_2, a_2, s_3) + \dots \quad (7.1)$$

Если число шагов конечное, то это **эпизодическая задача** и награды суммируются по эпизоду (до терминального состояния). Сокращенно это можно записать в виде

$$U([r_0, r_1, r_2, \dots]) = r_0 + r_1 + r_2 + \cdots, \quad (7.2)$$

где r_0, r_1,… – награды на каждом шаге. В моделях **конечного горизонта** эпизод прерывается после фиксированного числа шагов (аналогично ограничению глубины в деревьях поиска).

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

$$U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots. \quad (7.3)$$

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 ($\gamma<1$):

$$U([r_0, \dots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \le R_{\max}/(1-\gamma). \quad (7.4)$$

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*:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t \ge 0} \gamma^t r_t | s_0 = s, \pi\right], \quad (7.5)$$

где $\mathbb{E}$ – символ математического ожидания.

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

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0}\gamma^t r_t | s_0 = s, a_0 = a, \pi\right]. \quad (7.6)$$

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

Мы хотим найти **оптимальную политику π***, которая максимизирует накопленную награду. *Формально оптимальная политика определяется выражением*

$$\pi^* = \arg\max_{\pi}\mathbb{E}\left[\sum_{t\geq 0}\gamma^t r_t | \pi\right]. \quad (7.7)$$

7.2.3. MDP дерево поиска и уравнение Беллмана

Марковский процесс принятия решений, также как поиск в пространстве состояний, можно представить в виде дерева поиска (рисунок 7.1). Неопределенность моделируется в этих деревьях с помощью **Q-состояний**, также известных как узлы **состояния-действия** (*s*, *a*), по существу идентичных узлам жеребьевки в Expectimax деревьях; **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, выполнив действие *a*, перейдет в состояние *s*'.

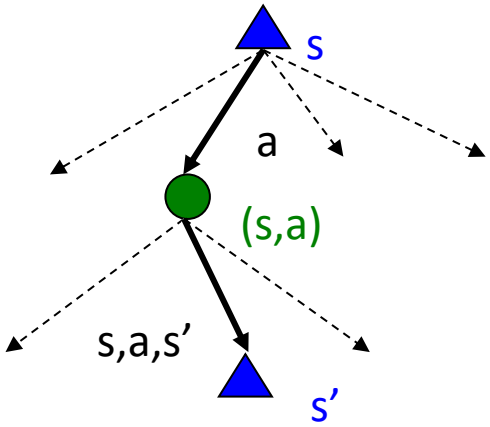


Рисунок 7.1. – MDP дерево поиска

Для решения задачи MDP используют уравнение Беллмана. Когда мы ищем решение задачи, представленной в виде MDP, интерес представляет нахождение оптимальных функций ценности и политики.

Оптимальной функцией ценности $V^*(s)$ называется функция ценности, которая обеспечивает максимальную ценность при следовании определенной политике [7]:

$$V^*(s) = \max_{\pi} V^{\pi}(s). \quad (7.8)$$

Оптимальную функцию ценности состояния вычисляют как максимум *Q*-функции по действиям *a*:

$$V^*(s) = \max_a Q^*(s, a). \quad (7.9)$$

Ценность *q*-состояния (*s*,*a*) можно определить выражением (см. рисунок 7.1):

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[\, R(s,a,s') + \gamma\, V^*(s')\,] \quad . \quad (7.10)$$

Подставив (7.10) в (7.9), получим **уравнение Беллмана**

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \quad , \quad (7.11)$$

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.

7.2.4. Итерации по значениям ценности состояний

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

$$\forall s, V_{k+1}(s) = V_k(s)). \quad (7.12)$$

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

$$\forall s \in S,\ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')] \quad . \quad (7.13)$$

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 задолго до того, как сойдутся ценности состояний.

7.2.5. Извлечение 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:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \quad . \qquad (7.14)$$

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

$$\pi^*(s) = \arg\max_a Q^*(s,a) \quad . \qquad (7.15)$$

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

7.2.6. 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:

$$V^{\pi}(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^{\pi}(s')] \quad . \qquad (7.16)$$

   Вычисление $V^{\pi}(s)$ можно выполнить *двумя способами*: либо решить систему линейных уравнений (7.16) относительно $V^{\pi}(s)$ (например, в Matlab); либо вычислить оценки состояний итерационно, используя пошаговые обновления:

$$V_0^{\pi}(s) = 0,$$

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')] \quad . \qquad (7.17)$$

   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**:**

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi_i}(s') \right] . \quad (7.18)$$

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

Важно отметить, что политика в этом случае сходится к оптимальной политике. Кроме этого, схождение политики происходит быстрее схождения значений ценности состояний.

7.2.7. Обучение с подкреплением

Обучение с подкреплением (**RL**, reinforcement learning) — область машинного обучения, в которой обучение осуществляется посредством взаимодействия агента с окружающей средой (рисунок 7.2) [7, 8]. В среде RL вы не указываете агенту, что и как он должен делать, вместо этого **агент получает награду за каждое выполненное действие**. В итоге агент начинает выполнять действия, которые максимизируют вознаграждение.

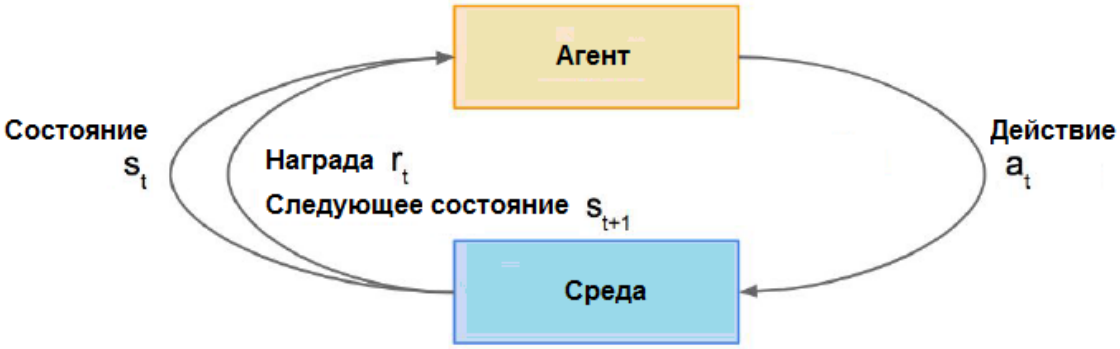


Рисунок 7.2 – Обучение с подкреплением

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 обучением на основе подкрепления, прежде, чем начнет **эксплуатировать** (**expoloitation**) 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**.

7.2.8. 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:**

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

Шаг 2: **Получение решения MDP:**

После схождения оценок T, обучение завершается генерацией политики $\pi(s)$ 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')).

7.2.9. Обучение 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.

7.2.10. Алгоритм прямого оценивания (обучение без модели)

Цель: вычисление ценности каждого состояния при следовании политике π.

Идея: Усреднять наблюдаемые выборки значений ценности.

Алгоритм:

- Действовать в соответствии с политикой π:
  - каждый раз при посещении состояния, подсчитывать и аккумулировать ценности состояния и число посещений состояния;
  - найти среднюю ценность состояния.

Положительные свойства прямого оценивания: простота; не требует знаний *T, R*; вычисляет средние значения ценностей, используя просто выборки переходов.

Основным недостатком алгоритма являются значительные временные затраты.

7.2.11. Обучение на основе временных различий (TD-обучение)

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 $V(s)$:

- выборочное значение $V(s)$:

$$sample = R(s,\pi(s),s') + \gamma V^{\pi}(s') \tag{7.19}$$

- скользящее среднее выборок:

$$V^{\pi}(s) \leftarrow (1 - \alpha)V^{\pi}(s) + (\alpha)\, sample \tag{7.20}$$

или

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\,(sample - V^{\pi}(s)). \tag{7.21}$$

Правило обновления ценности состояния (7.21) 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 шагов **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 (7.20);
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.

7.2.12. Q-обучение

TD-обучение – подход к оцениванию политики без модели, моделирующий обновление Беллмана с помощью он-лайн усреднения выборок. Однако, если необ-

ходимо будет преобразовать значения $V^{\pi}(s)$ в новую политику возникнут сложности, так как для этого в соответствии с (7.15) необходимо оперировать значениями *q*-ценностей.

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

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]. \quad (7.22)$$

Это правило обновления следует из (7.10) и (7.9). Алгоритм q-обучения строится по схеме, аналогичной алгоритму TD-обучения, с использованием *выборок значений Q-состояний*

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a') \quad (7.23)$$

и их скользящем усреднении

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot sample. \quad (7.24)$$

или

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

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

При достаточном объеме исследований и снижении в ходе обучения скорости обучения α оценки *Q*-ценностей, получаемые с помощью (7.24), 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.

7.12.13. Алгоритм двойного Q-обучения

Q-обучение может завышать оценки ценности действий, главным образом из-за компоненты $\max_a Q(s', a')$. Алгоритм двойного *Q*-обучения призван преодолеть этот недостаток посредством использования двух *Q*-функций, которые обозначим *Q1* и *Q2*. На каждом шаге обновляется одна функция. Если выбрана *Q1*, то обновление производится по формуле:

$$a^* = \operatorname{argmax}_a Q1(s', a'); \quad (7.25)$$

$$Q1(s,a) \leftarrow Q1(s,a) + \alpha[R(s,a,s') + \gamma Q2(s', a^*) - Q1(s,a)], \quad (7.26)$$

а если *Q2*, то по формуле

$$a^* = \operatorname{argmax}_a Q2(s', a'); \quad (7.27)$$

$$Q2(s,a) \leftarrow Q2(s,a) + \alpha[R(s,a,s') + \gamma Q1(s', a^*) - Q2(s,a)] \quad (7.28)$$

Это значит, что в обновлении каждой Q-функции участвует другая функция. При этом применяется жадный поиск, что уменьшает степень завышения оценки ценности действий по сравнению с одной Q-функцией.

7.2.14. ε-жадная стратегия

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 среди уже исследованных на основе текущей политики с вероятностью 1 – ε 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.

7.2.15. Q-обучение с линейной аппроксимацией

Обычное Q-обучение требует вычисления значений *Q(s, a)* по всем возможным парам состояние-действие. Но представьте среду, в которой число состояний очень велико, а в каждом состоянии доступно множество действий. Перебор всех действий в каждом состоянии занял бы слишком много времени.

Возможное решение – аппроксимировать функцию *Q*(*s*, *a*). Для этого представляют ценность состояний с помощью *вектора признаков.* **Признаки** - это функции, которые фиксируют важные свойства состояний. Примеры признаков состояний для игры Пакман:

- расстояние до ближайшего призрака;
- расстояние до ближайшей гранулы;
- число призраков;
- 1/ (расстояние до призрака)^2;
- находится ли Пакман в ловушке? (0/1);
- и др..

Используя вектор признаков, мы можем аппроксимировать Q-функцию ценности состояния, например, в виде линейной комбинации признаков с весами:

$$Q(s,a) \quad = \quad w_1 \cdot f_1(s,a) + w_2 \cdot f_2(s,a) + \ldots + w_n \cdot f_n(s,a) = \vec{w} \cdot \vec{f}(s,a) \quad (7.29)$$

где $\vec{f}(s,a) = \begin{bmatrix} f_1(s,a) & f_2(s,a) & \ldots & f_n(s,a) \end{bmatrix}^T$ – вектор признаков состояния (*s*, *a*), а $\vec{w} = \begin{bmatrix} w_1 & w_2 & \ldots & w_n \end{bmatrix}$ – вектор весов.

Основные шаги **алгоритма Q-обучения с аппроксимацией**:

1) выполнить действие в состоянии *s* в соответствии с ε-жадной стратегией и получить выборку (*s*, *a*, *s'*, *r*);

2) вычислить разность (*difference*) между выборочным значением *Q*-ценности $[R(s,a,s') + \gamma \max_{a'} Q(s',a')]$ и текущим значением $Q(s, a)$

$$difference = [R(s,a,s') + \gamma \max_{a'} Q(s',a')] - Q(s,a) :$$

3) обновить веса признаков состояний в соответствии с простым *дельта правилом*

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a). \quad (7.30)$$

и вычислить новые значения ценности $Q(s, a)$ в соответствии с (7.29);

4) повторять шаги 1)-3) до схождения Q-ценностей.

Вместо того, чтобы хранить Q-ценности для каждого состояния **Q-обучение с аппроксимацией** позволяет хранить только один весовой вектор. В результате это даёт более эффективную версию алгоритма с точки зрения использования памяти.

7.2.16. Функция разведки (исследования)

При использовании рассмотренного алгоритма *q*-обучения с аппроксимацией требуется вручную управлять коэффициентом ε. Этого можно избежать, если применить **функцию разведки (исследования)**, которая использует модифицированное правило обновления ценности *q*-состояний, предоставляющее предпочтения тем состояниям, которые посещаются реже:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} f(s',a')], \quad (7.31)$$

где f – функция разведки. Обычно полагают, что

$$f(s,a) = Q(s,a) + \frac{k}{N(s,a)}, \quad (7.32)$$

где k – некоторая предопределенная константа, $N(s,a)$ – число посещений *q*-состояния (s, a). Агент в состоянии s всегда выбирает действие с максимальным значением $f(s,a)$ и, соответственно, не принимает вероятностных решений относительно исследований и эксплуатации, так как решение в пользу исследования новых выборок автоматически регулируется в (7.32) членом $k/N(s, a)$, который будет иметь большие значения для редко выполняемых действий. По мере увеличения числа итераций, счетчики числа посещений состояний растут и $k/N(s, a)$ стремится к нулю, а $f(s,a)$ стремится к $Q(s,a)$. Таким образом, исследования выборок становятся все более и более редкими.

7.2.17. Алгоритм двойной глубокой *Q*-сети (DDQN)

Если в качестве аппроксиматора $Q(s, a)$ используется глубокая нейронная сеть (DQN – deep *Q*-network), то говорят **о глубоком *Q*-обучении**[8]. В **алгоритме двойной DQN** (Double DQN) для оценивания целевых *Q*-значений используется

отдельная сеть, которая имеет такую же структуру, как сеть, выполняющая аппроксимацию $Q(s, a)$. Но ее веса изменяются только после каждых T эпизодов (T – настраиваемый гиперпараметр). Обновление сводится просто к копированию весов аппроксимирующей (предсказательной) сети. Таким образом, целевая Q-функция в течение некоторого времени остается неизменной, что повышает устойчивость процесса обучения.

Математически обучение двойной DQN означает минимизацию в ходе обучения функции потерь в виде среднего квадрата следующей ошибки:

$$difference=[\ R(s,a,s')+\gamma\ \max_a Q_T(s',a')\ ] - Q(s,a), \qquad (7.33)$$

где $Q_T(s',a')$ – функция ценности целевой сети, а $Q(s,a)$ – функция ценности предсказательной сети.

Соответственно, правило обновления весов нейросети в этом случае будет отличаться от (7.30) и базироваться на алгоритме обратного распространения ошибки, используемого при обучении нейронных сетей.

7.2.18. DQN с буфером воспроизведения опыта

Аппроксимация значений Q-функций нейронной сетью, обучаемой на одном примере за раз, ведет себя не устойчиво. Для повышения устойчивости используют **буфер воспроизведения опыта**.

Идея заключается в том, чтобы сохранять в памяти опыт агента в виде выборок (s, a, s', r), полученных на протяжении эпизодов в сеансе обучения. Обучение с буфером воспроизведения опыта состоит из 2-х этапов: накопление опыта и обновление модели (ей) на миниблоке из случайных выборок прошлого опыта.

Воспроизведение опыта может стабилизировать процесс обучения, обеспечив набор слабо коррелированных примеров.

7.3. Задания для выполнения

Задание 1. Итерации по значениям

Реализуйте агента, осуществляющего итерации по значениям в соответствии с выражением (7.13), 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).

Подсказка: в среде BookGrid, используемой по умолчанию, при выполнении 5 итераций по значениям должен получиться результат, изображенный на рисунке 7.3:

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

Оценивание: ваш агент, использующий итерации по значениям, будет оцениваться с использованием иной схемы клеточного мира. Будут проверяться ценности состояний, *q*-ценности и политики после заданного количества итераций, а также с учетом выполнения условия сходимости (например, после 100 итераций).

Задание 2. Реализация политик

Рассмотрим схему среды **DiscountGrid**, изображенную на рисунке 7.4. 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» вдоль нижней строки схемы, эти пути короче, но характеризуются большими отрицательными наградами (они

обозначены красной стрелкой на рисунке 7.4); (2) пути, которые «избегают обрыва» и проходят по верхней строке схемы. Эти пути длиннее, но они с меньшей вероятностью принесут отрицательные результаты. Эти пути обозначены зеленой стрелкой на рисунке 7.4.

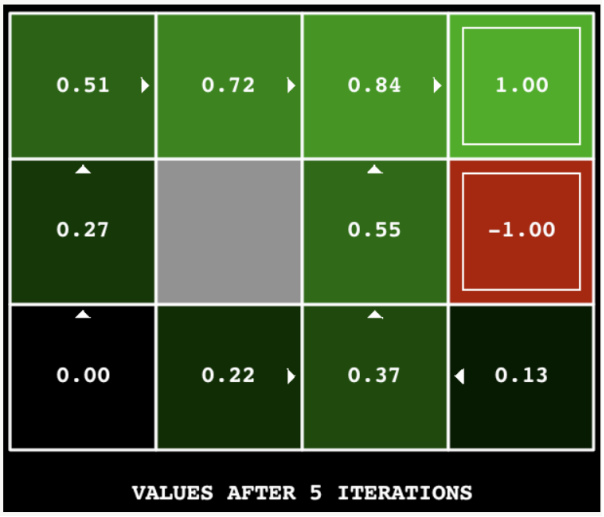


Рисунок 7.3 – Состояние среды BookGrid после 5 итераций

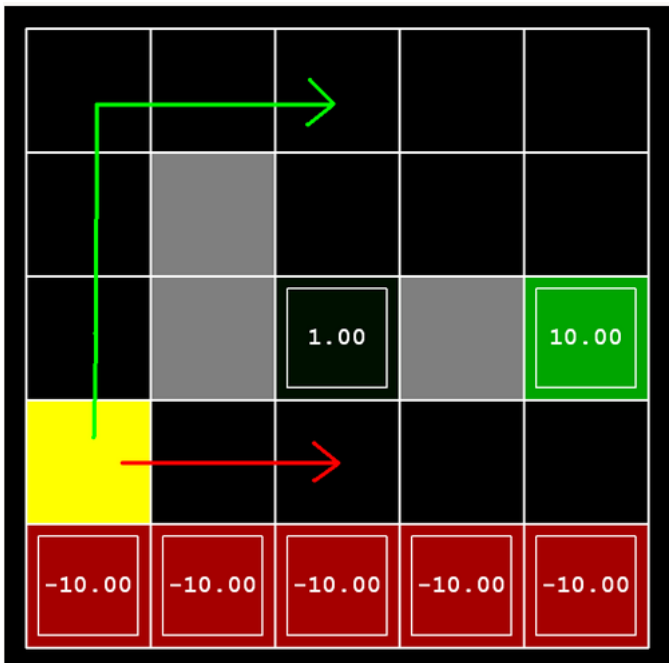


Рисунок 7.4 – Состояния среды DiscountGrid

В этом задании необходимо подобрать значения коэффициента дисконтирования, уровня шума и текущей награды для схемы **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), перемещаясь вдоль обрыва (-10);
2) агент предпочитает близкий выход (+1), но избегает обрыва (-10);
3) агент предпочитает дальний выход (+10), перемещаясь 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 **question2a()** до **question2e()** в файле **analysis.py**. Эти функции возвращают кортеж из трех элементов (дисконт, шум, награда).

Задание 3. Q-обучение

Обратите внимание, что ваш агент итераций по значениям фактически не обучается на собственном опыте. Он скорее, изучает 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 Gridword, 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, 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 6, когда вы переопределите **getQValue** для работы с признаками пар «состояние-действие».

Задание 4. Эпсилон-жадная стратегия

Завершите реализацию агента с *q*-обучением, дописав эпсилон-жадную стратегию выбора действий в методе **getAction** класса **QLearningAgent**.Метод должен обеспечивать выбор случайного действия с вероятностью **epsilon** и с вероятностью

1- epsilon 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 субоптимальное действие, а любое случайное допустимое действие.

Вы можете выбрать действие из списка случайным образом, вызвав функцию **random.choice**. Для моделирования двоичной случайной переменной используйте вызов **util.flipCoin(epsilon)**, который возвращает **True** с вероятностью **epsilon** и **False** с вероятностью **1- epsilon**.

Задание 5. Q-обучение и Пакман

Пора поиграть в Пакман! Игра будет проводиться в два этапа. На первом этапе обучения Пакман будет 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 установлены нулевые значения, что фактически остановит *q*-обучение и отключит режим исследования, чтобы позволить Пакману использовать сформированную входе обучения политику. Тестовая игра отображается в графическом интерфейсе. Запустите *q*-обучение Пакмана для маленького поля игры **smallGrid**:

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

Обратите внимание, что **PacmanQAgent** уже определен для вас в терминах 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 этой части задания, если агент будет выигрывать не менее, чем в 80% случаев. Автооцениватель проведет 10 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, могли иметь отрицательные значения *q*-ценности, то ненаблюдаемое действие может стать оптимальным. Остерегайтесь применения функции argmax класса **util.Counter.**

Чтобы оценить задание 5, выполните:

```
python autograder.py -q q5
```

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 агенту как **self.epsilon**, **self.gamma** и **self.alpha**.

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, как Pacman играет только в последние 10 игр. Количество обучающих игр может передаваться вашему агенту также в виде опции **numTraining**.

Статистика обучения будет отображаться после каждых 100 игр. Так как параметр **epsilon** имеет положительные значения во время обучения, то Пакман играет плохо даже после того, как усвоит хорошую политику. Это происходит потому, что он иногда совершает 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).

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

Тем не менее вы обнаружите, что обучение того же агента, казалось бы, на простой среде **mediumGrid** не работает. При этом среднее вознаграждение Пакмана остается отрицательным на протяжении всего обучения. Во время теста он играет плохо, вероятно, проигрывая все свои тестовые партии. Тренировки также займут много времени.

Пакман не может победить на больших игровых полях, потому что каждая конфигурация игры имеет специфические состояния с уникальными 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.

Задание 6. Q-обучение с аппроксимацией

Реализуйте *q*-обучение с аппроксимацией, которое обеспечивает обучение весов признаков состояний. Дополните описания 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 $[f1(s, a), \ldots fi(s, a), \ldots, fn(s, a)]$ из значений признаков. Вам предоставляются для этого возможности модуля **featureExtractors.py.** Вектор признаков – это объект **util.Counter** (подобен словарю), содержащий пары признаков и значений; все пропущенные признаки будут иметь нулевые значения.

Задание 7. Глубокое Q-обучение

В этом итоговом задании объединяются идеи Q-обучения и машинного обучения из предыдущей лабораторной работы. Вам необходимо в файле **model.py** реализовать класс **DeepQNetwork**, который представляет собой глубокую нейронную сеть, предсказывающую Q-ценности для всех возможных пар (s, a). В ходе выполнения задания вам необходимо определить конструктор нейросети **__init__** , реализовать её методы прямого распространения **forward,** вычисления потерь **get_loss,** одношагового обучения **gradient_update**.

7.4. Порядок выполнения лабораторной работы

7.4.1. Изучить по лекционному материалу и учебным пособиям [1-3, 7] методы недетерминированного поиска, основанные на использовании модели марковского процесса приятия 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.

7.4.2. Использовать для выполнения лабораторной работы файлы из архива **МиСИИ_лаб7_2024.zip.** Разверните программный код в новой папке и не смешивайте с файлами предыдущих лабораторных работ.

7.4.3. В этой лабораторной работе необходимо реализовать алгоритмы итераций по значениям и Q-обучение, включая глубокое Q-обучение.

Для автооценивания всех заданий лабораторной работы следует выполнить команду:

```
python autograder.py
```

Для оценки конкретного задания, например, задания 2, автооцениватель 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
```

Код лабораторной работы содержит файлы, указанные в таблице ниже.

Файлы для редактирования:	
valueIterationAgents.py	Агенты, выполняющие итерацию по значениям для решения известного MDP.
qlearningAgents.py	Агенты, использующие Q-обучения, для классов Gridworld, Crawler и Pacman.
analysis.py	Файл для размещения Ваших ответов на вопросы заданий лабораторной работы.
model.py	Файл, содержащий класс DeepQNetwork
Файлы, которые необходимо просмотреть:	

mdp.py	Определяет методы для общих MDP.
learningAgents.py	Определяет базовые классы ValueEstimationAgent и QLearningAgent, которые Ваши агенты должны расширить.
util.py	Утилиты, включающие util.Counter, которые полезны для Q-обучения.
gridworld.py	Реализация класса клеточного мира - Gridworld.
featureExtractors.py	Классы для извлечения признаков пар (состояние, действие). Используются в Q-обучении с аппроксимацией (в qlearningAgents.py).
Поддерживающие файлы, которые можно игнорировать:	
environment.py	Абстрактный класс для общей среды обучения с подкреплением. Используется в gridworld.py.
graphicsGridworldDisplay.py	Gridworld графика.
graphicsUtils.py	Графические утилиты.
textGridworldDisplay.py	Плагин для текстового интерфейса Gridworld.
crawler.py	Код робота Crawler и тест-комплект.
graphicsCrawlerDisplay.py	GUI для краулера.
autograder.py	Автоооцениватель для лабораторной работы
testParser.py	Парсер тестов автооценивателя и файлы решений
testClasses.py	Общие классы автооценивателя
test_cases/	Папка, содержащая тесты для каждого из заданий (вопросов)
reinforcementTestClasses.py	Особые тестовые классы авооценивателя

7.4.4. 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, 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.

Примечание: среда **Gridworld** такова, что агент сначала должен 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 Pacman, позиции представлены декартовыми координатами **(x, y),** а любые массивы индексируются **[x] [y]**, где «север» является направлением увеличения **y** . По умолчанию большинство переходов получают нулевую награду. Это можно изменить с помощью опции **–r**, которая управляет текущей наградой.

7.4.5. В задании 1 требуется реализовать следующие методы класса **ValueIterationAgent(ValueEstimationAgent):**

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

Метод **runValueIteration(self)** в циклах для каждой итерации и для каждого состояния **state** осуществляет выбор действий в состоянии (с помощью **action in self.mdp.getPossibleActions(state)**) и для каждой пары (**state, action**) вычисляет значения *q*-ценностей и складирует их в некотором списке **q_state** с помощью вызова **q_state.append(self.getQValue(state, action))**. Вычисление ценности каждого состояния реализуется на основе (7.9) путем вызова **updatedValues[state] = max(q_state)**, где **updatedValues** – временный словарь, содержащий обновляемые ценности состояний **state** на каждой итерации. После вычисления ценности всех состояний для заданной итерации, они сохраняются в словаре значений ценности состояний путем копирования **self.values = updatedValues**.

Метод **computeQValueFromValues(self, state, action)** вычисляет ценности *q*-состояний в соответствии с (7.10). Для каждого следующего состояния **nextstate** после **state** вычисления реализуются с помощью вызова:

```
Qvalue+=prob*(self.mdp.getReward(state, action, nextstate) + self.dis-
count*self.getValue(nextstate)),
```

где **nextstate** и **prob** извлекаются с помощью **in** из 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 (7.15).

Протестируйте вашу реализацию и внесите результаты тестирования в отчет:

python autograder.py -q q1

7.4.6. В задании 2 необходимо подобрать значения коэффициента дисконтирования (**answerDiscount**), уровня шума (**answerNoise**) и текущей награды (**answerLivingReward**) для среды **DiscountGrid** (рисунок 7.4). Внесите вместо **None** необходимые значения параметров в функции от **question2a()** до **question2e()** в файле **analysis.py**.

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.

Протестируйте ваши ответы и внесите результаты тестирования в отчет:

python autograder.py -q q2

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

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

7.4.7. В задании 3 необходимо реализовать для класса **QLearningAgent** 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 вычисления лучшей ценности **bestQ** состояния **state**. Затем в цикле перебираются все допустимые действия для **state**, вычисляются с помощью **getQValue(state, action)** ценности q-состояний и находятся действия **action** с ценностью, равной **bestQ.** Найденные лучшие действия накапливаются в некотором списке. Метод осуществляет случайный выбор лучшего действия среди найденных лучщих действий.

В методе **update(self, state, action, nextState, reward)** необходимо реализовать q-обучение в соответствии с (7.23) и (7.24). Для вычисления ценности состояния s' – **nextState** используйте вызов **getValue(nextState)**, который реализует (7.9).

Можно 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.

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 Pacman на север, а затем на восток по оптимальному пути для четырех эпизодов, вы должны увидеть значения *q*-ценностей, указанные на рисунке 7.5.

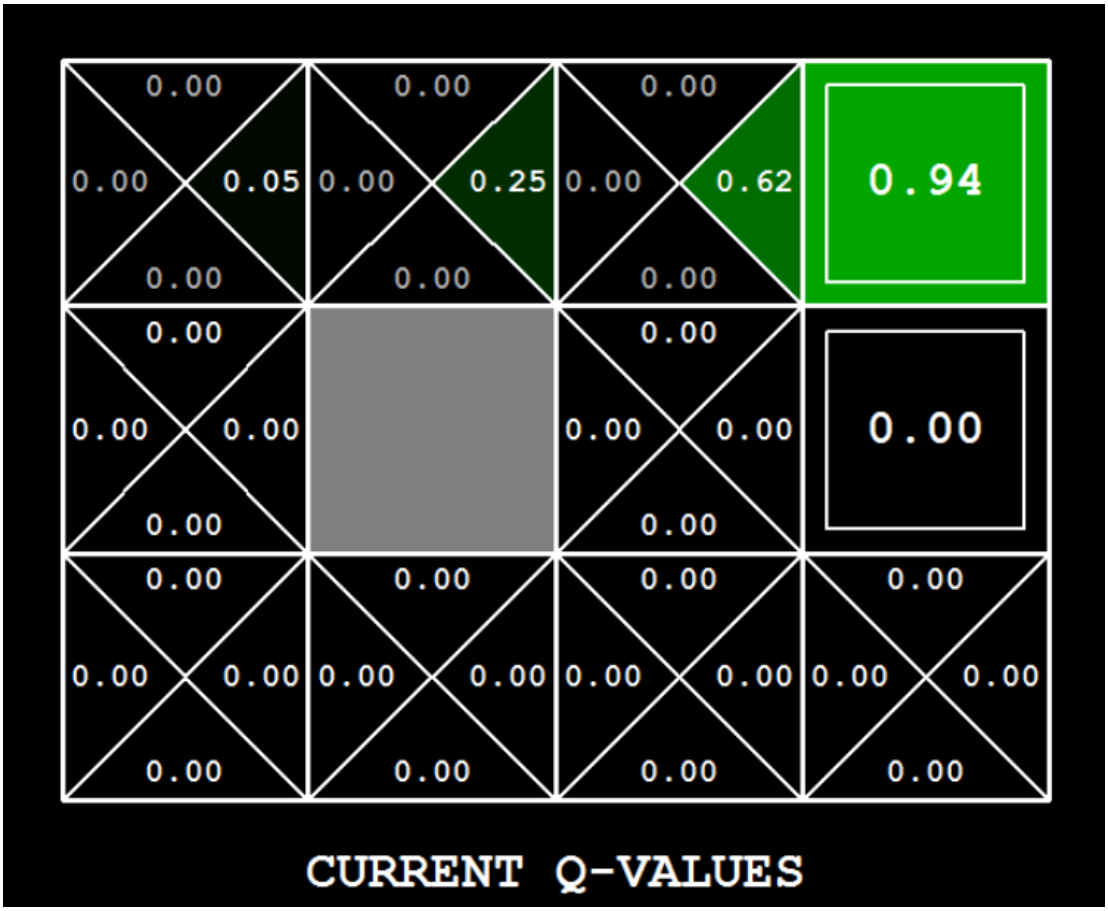


Рисунок 7.5. – Результаты *q*-обучения

В ходе автоценивания будет проверяться, обучается ли агент тем же значениям *q*-ценностей и политике, что и эталонная реализация на одном и тот же наборе примеров. Чтобы оценить вашу реализацию, запустите автооцениватель и внесите результаты в отчет:

python autograder.py -q q3

7.4.8. В задании 4 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).**

После реализации метода **getAction** проанализируйте поведение агента в **gridworld** (с **epsilon = 0.3**).

python gridworld.py -a q -k 100

Ваши окончательные значения *q*-ценностей должны будут 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.

Проанализируйте 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 автооцениватель внесите результаты в отчет:

python autograder.py -q q4

Теперь без дополнительного кодирования вы сможете запустить *q*-обучение для робота **crawler** (рисунок 7.6**)**:

python crawler.py

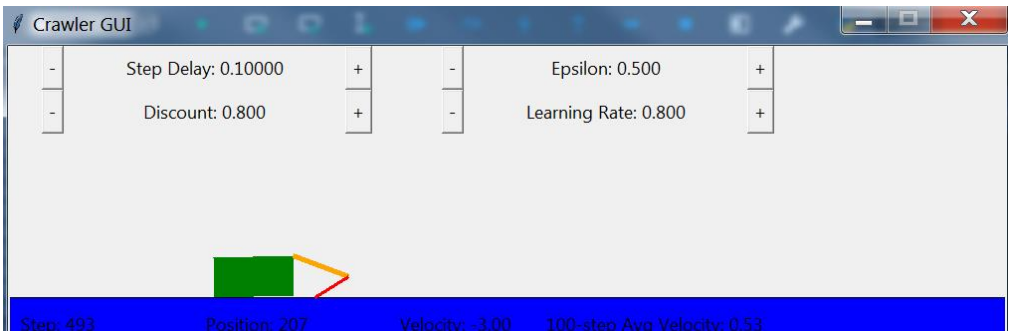


Рисунок 7.6. – Робот Crawler

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**. Испытайте различные параметры 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.

7.4.9. При выполнении задания 5 следуйте указаниям, приведенным в самом задании. Проведите все необходимые эксперименты, указанные в задании. Результаты внесите в отчет.

7.4.10. В задании 6 необходимо реализовать *q*-обучение с аппроксимацией, дописав методы класса **ApproximateQAgent** .

При написании кода метода **getQValue(self, state, action)**, возращающего 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 (7.25).

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 (7.26).

По умолчанию **ApproximateQAgent** использует функцию **IdentityExtractor**, которая для каждой пары **(state, action)** – состояние-действие создает отдельный признак. С такой функцией извлечения признаков агент, осуществляющий *q*-обучение с аппроксимацией, будет работать аналогично **PacmanQAgent**. Проверьте 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

SimpleExtractor возвращает простые признаки:

- есть ли возможность съесть еду в следующей позиции;
- как далеко находится следующая еда;
- неизбежно ли столкновение с призраком;
- находится ли призрак в шаге от агента.

Даже более сложные игровые поля не должны стать проблемой для **ApproximateQAgent** с **SimpleExtractor** (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.

В ходе оценивания будет проверяться, обучается ли ваш агень тем же значениям *q*-ценностей и весам признаков, что и эталонная реализация. Чтобы оценить вашу реализацию, запустите автооцениватель и внесите результаты всех экспериментов в отчет:

```
python autograder.py -q q6
```

7.4.11. В задании 6 необходимо в файле **model.py** реализовать в классе **DeepQNetwork** глубокую нейронную сеть, вычисляющую целевые значения Q-ценностей для всех возможных пар (*s, a*), и использовать её в алгоритмах двойного Q-обучения или DDQN, рассмотренных выше.

При реализации контсруктора класса **DeepQNetwork** в методе **__init__()** инициализируйте следующие обязательные параметры и переменные:

self.learning_rate: скорость обучения нейросети, которая будет использоваться оптимизатором;

self.numTrainingGames: число обучающих игр, которые будут использоваться, чтобы собрать выборки (*s, a, s', r*) и провести Q-обучение; обратите внимание, что это число должно быть больше 1000, так как примерно первые 1000 игр используются для исследования и не используются для обновления Q-ценностей.

self.batch_size: размер миниблока данных из буфера воспроизведения опыта, который используется на каждом шаге обновления параметров нейросети на основе алгоритма обратного распространения; этот параметр будет использовать автооценивателем, вам не нужно будет обращаться к нему непрсоредственно.

Определите также следующие методы класса **DeepQNetwork:**

forward(self, states): возвращает результат прямого распространения миниблока данных **states** размером **(batch_size, state_dim)** по нейросети, где **state_dim** – размерность состояния игры; метод возвращает тензор размером **(batch_size, num_actions)**, который содержит прогнозируемые нейросетью Q значения для всех возможных действий в каждом состоянии из **states**;

get_loss(self, states, Q_target): возвращает средний квадрат ошибки между прогнозируемыми нейросетью значениями Q, которые возвращает метод **forward()** и целевыми значениями **Q_targets** (которые рассматриваются как истинные);

gradient_update(): выполняет одну итерацию обновления параметров нейросети **self.parameters()** с использованием одного из оптимизаторов машинного обучения, например, **SGD, Adam**; метод должен выполнять только одно обновление каждого параметра, т.к. цикличность обновления будет обеспечиваться автооценивателем. Чтобы не создавались копии оптимизаторов при каждом вызове **gradient_update(),** рекомендуется не включать вызов конструктора оптимизатора в код метода **gradient_update()**, вместо этого разумно определить оптимизатор в конструкторе класса **DeepQNetwork.** В таком случае экземпляр оптимизатора будет создавться один раз при инициалиации **DeepQNetwork.**

Непосредственно алгоритмы глубокого Q обучения Пакмана реализованы в классе **PacmanDeepQAgent**. Класс позволяет реализовать алгоритм двойного Q-обучения (см. п. 7.2.13, параметр **doubleQ=True**) или двойного DQN (см. п. 7.2.17, параметр **doubleQ=False**) с использованием буфера воспроизведения опыта (см. п. 7.2.18). Просмотрите код класса, который содержит комментарии для лучшего понимания.

Поскольку вам необходимо разработать подходящую архитектуру глубокой нейросети, то познакомьтесь с методом get_features, который возвращает вектор, описывающий состояние игры state. Видно, что состояние игры представляется вектором, содержащим позицию Пакмана (**pacman_state**), позиции приведений (**ghost_state**), позиции капсул с едой (**food_locations**). Размерность этого вектора вычисляется при вызове метода **get_state_dim()**, который вернет размер входа нейросети **state_dim**. В ходе обучения глубокая нейросеть должна по сути по входным состояниям игры научиться формировать скрытые признаки, подходящие для аппроксимации Q-функции. Ваша задача подобрать архитектуру сети, обеспечивающую удовлетворительное прогнозирование значений Q(*s, a*). Начните с простой сети, содержащей один скрытый слой, и постепенно её усложняйте.

После завершения разработки сети выполните автооценивание. В ходе автооценивания агент **DeepQPacman** будет тестироваться на 10 играх после того, как он обучится на **self.numTrainingGames** играх. Если агент выиграет не менее 6 из 10 игр, то тест бужет пройден. Обратите внимание, что алгоритмы DQN не отличается стабильностью. Количество игр, в которых выиграет агент, может меняться для каждого запуска.

Запустите автооцениватель следующей командой и внесите результаты в отчет:

python autograder.py -q q7

7.5. Содержание отчета

Цель работы, описание основных понятий MDP, уравнение Беллмана, описание алгоритмов итераций по значениям и политикам, описание задачи обучения с подкреплением, описание алгоритма RL, основанного на модели, описание алгоритмов TD-обучения и Q-обучения, описание алгоритма Q-обучения с аппроксимацией, алгоритмы двойного Q-обучения и DDQN, код реализованных агентов и функций с комментариями в соответствии с заданиями 1-7, результаты игр на разных полях игры, их анализ, результаты автооценивания заданий, выводы по проведенным экспериментам с разными алгоритмами обучения с подкреплением.

7.6. Контрольные вопросы

7.6.1 Объясните, что понимают под недетерминированными задачами поиска?

7.6.2. Объясните основные понятия Марковского процесса принятия решений.

7.6.3. Что понимается под функцией политики?

7.6.4. Что понимается под функцией полезности?

7.6.5. Какая задача называется эпизодической?

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

7.6.8. Как определяется функция ценности состояния?

7.6.9. Как определяется *q*-функция ценности состояния-действия?

7.6.10. Объясните, как строится MDP дерево поиска?

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

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

7.6.13. Сформулируйте и объясните метод извлечения политики.

7.6.14. Сформулируйте и объясните алгоритм итерации по политикам.

7.6.15. Объясните основные понятия обучения с подкреплением.

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

7.6.17. Объясните обучение с подкреплением на основе модели.

7.6.18. Как классифицируются алгоритмы обучения с подкреплением без модели?

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

7.6.20. Объясните алгоритм обучения на основе временных различий.

7.6.21. Объясните алгоритм *Q*-обучения.

7.6.22. Объясните, что понимают под ε-жадная стратегией.

7.6.23. Объясните алгоритм *Q*-обучения с аппроксимацией.

7.6.24. Что понимают под функции разведки (исследования)?

7.6.25. Объясните алгоритм двойного *Q*-обучения.

7.6.26. Объясните алгоритм двойной глубокой *Q*-сети.

7.6.27. Объясните механизм использования буфера воспризведения опыта.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Бондарев, В. Н. Искусственный интеллект : учеб. пособие / В. Н. Бондарев, Ф. Г. Аде.— Севастополь : СевНТУ, 2002. — 615с. — Текст : непосредственный.
2. Люгер, Дж. Искусственный интеллект: стратегии и методы решения сложных проблем, 4-е изд. : пер. с англ. — М. : Вильямс, 2003. — 864с. — Текст : непосредственный.
3. Рассел, С. Искусственный интеллект: современный подход, 2-е изд. / С. Рассел, П. Норвиг : пер. с англ. — М. : Вильямс, 2006. — 1408с.
4. Шалимов, П. Ю. Функциональное программирование : учеб. пособие / П. Ю. Шалимов. — Брянск : БГТУ, 2003.— 160с. — Текст : непосредственный.
5. Дейтел, П. Python : Искусственный интеллект, большие данные и облачные вычисления / Дейтел П., Дейтел Х. — СПб.: Питер, 2020. — 864 **с. —** Текст : непосредственный.
6. John DeNero, Dan Klein Teaching Introductory Artificial Intelligence with Pac-Man. – URL: https://www.researchgate.net/publication/228577256_Teaching_Introductory_Artificial_Intelligence_with_Pac-Man (дата обращения: 22.04.2024)). **—** Режим доступа: открытый. — Текст : электронный.
7. Richard S. Sutton and Andrew G. Barto Reinforcement learning. An introduction: second edition. — London : The MIT Press Cambridge, Massachusetts, 2020. — 526 p. — Text: direct.
8. Равичандиран, Судхарсан Глубокое обучение с подкреплением на Python. OpenAI Gym и TensorFlow для профи / Равичандиран С. — СПб. : Питер, 2020. — 320 с. — Текст : непосредственный.
9. Сукар Л. Э. Вероятностные графовые модели. Принципы и приложения / Сукар Л. Э. ; пер. с англ. А. В. Снастина. — М : ДМК Пресс, 2021. — 338 с. — Текст : непосредственный.
10. Мэрфи К. П. Вероятностное машинное обучение: введение / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2022. — 940 с. — Текст :непосредственный.
11. Пойнтер Ян Программируем с PyTorch: Создание приложений глубокого обучения. — СПб.: Питер, 2020. — 256 с. — Текст :непосредственный.

Заказ № _____ от « _____ » _____2025г. Тираж _____экз.
Изд-во СевГУ