

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
СевГУ
2024

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

Р е ц е н з е н т ы :

В. С. Чернега – канд. техн. наук, доцент кафедры «Информационные системы»

А. А. Брюховецкий – канд. техн. наук, доцент кафедры
«Информационные технологии и компьютерные системы»

Ответственный за выпуск:

И. П. Шумейко – канд. физ.-мат. наук, доцент,
заведующий кафедрой «Информационные 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, 2024. – 135 с. – Текст : электронный.

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 2024 г.

Методические указания рассмотрены и рекомендованы к изданию на заседании Учёного Совета Института информационных технологий, протокол № 05 от 18 апреля 2024 г.

© Бондарев В. Н., сост., 2024
© ФГАОУ ВО «Севастопольский
государственный университет», 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	4
ЛАБОРАТОРНАЯ РАБОТА № 1	
«ИССЛЕДОВАНИЕ БАЗОВЫХ ФУНКЦИЙ ЯЗЫКА PYTHON».....	5
ЛАБОРАТОРНАЯ РАБОТА № 2	
«ИССЛЕДОВАНИЕ НЕИНФОРМИРОВАННЫХ МЕТОДОВ ПОИСКА РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»	34
ЛАБОРАТОРНАЯ РАБОТА № 3	
«ИССЛЕДОВАНИЕ ИНФОРМИРОВАННЫХ МЕТОДОВ ПОИСКА РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»	48
ЛАБОРАТОРНАЯ РАБОТА № 4	
«ИССЛЕДОВАНИЕ МЕТОДОВ МУЛЬТИАГЕНТНОГО ПОИСКА».....	60
ЛАБОРАТОРНАЯ РАБОТА № 5	
«ИССЛЕДОВАНИЕ МЕТОДОВ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ»....	76
ЛАБОРАТОРНАЯ РАБОТА № 6	
«ИССЛЕДОВАНИЕ ДИНАМИЧЕСКИХ СЕТЕЙ БАЙЕСА».....	108
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	135

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 проектирования интеллектуальных компьютерных систем. Основное внимание уделяется алгоритмам поиска решений задач в пространстве состояний, алгоритмам эвристического поиска, алгоритмам мультиагентного поиска, вероятностному выводу и алгоритмам обучения с подкреплением.

При выполнении лабораторных работ используется игровая среда AI Расмап, разработанная Джоном ДеНеро и Дэном Кляйном специально для обучения искусственному интеллекту [6]. Оригинальная видео игра Рас-Мап была разработана японской компанией Namco в 1980 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 неопределённых 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.

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 3.6. ***Предоставляемый код ряда модулей лабораторных работ не поддерживается в версиях выше Python 3.8+.***

Программу установки Python 3.6 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 версия выше, чем Python 3.8+, то рекомендуется создать отдельную среду для работы с Python 3.6, введя в командном окне conda команду:

```
conda create --name <env-name> python=3.6
```

Здесь <env-name> - название среды, например, `python36`. Чтобы войти в среду, которая была только что создана, активируйте её:

```
conda activate python36
```

Проверьте используемую в среде версию Python:

```
python -V
Python 3.6.13 :: Anaconda, Inc.
```

Установите в созданную среду Spider IDE командой:

```
conda install spyder
```

Среда готова для работы. Для выхода из среды `python36` 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 в архиве: `МиСИИ_лаб1.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, изображенное 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 рисунке 1.1 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 «!».

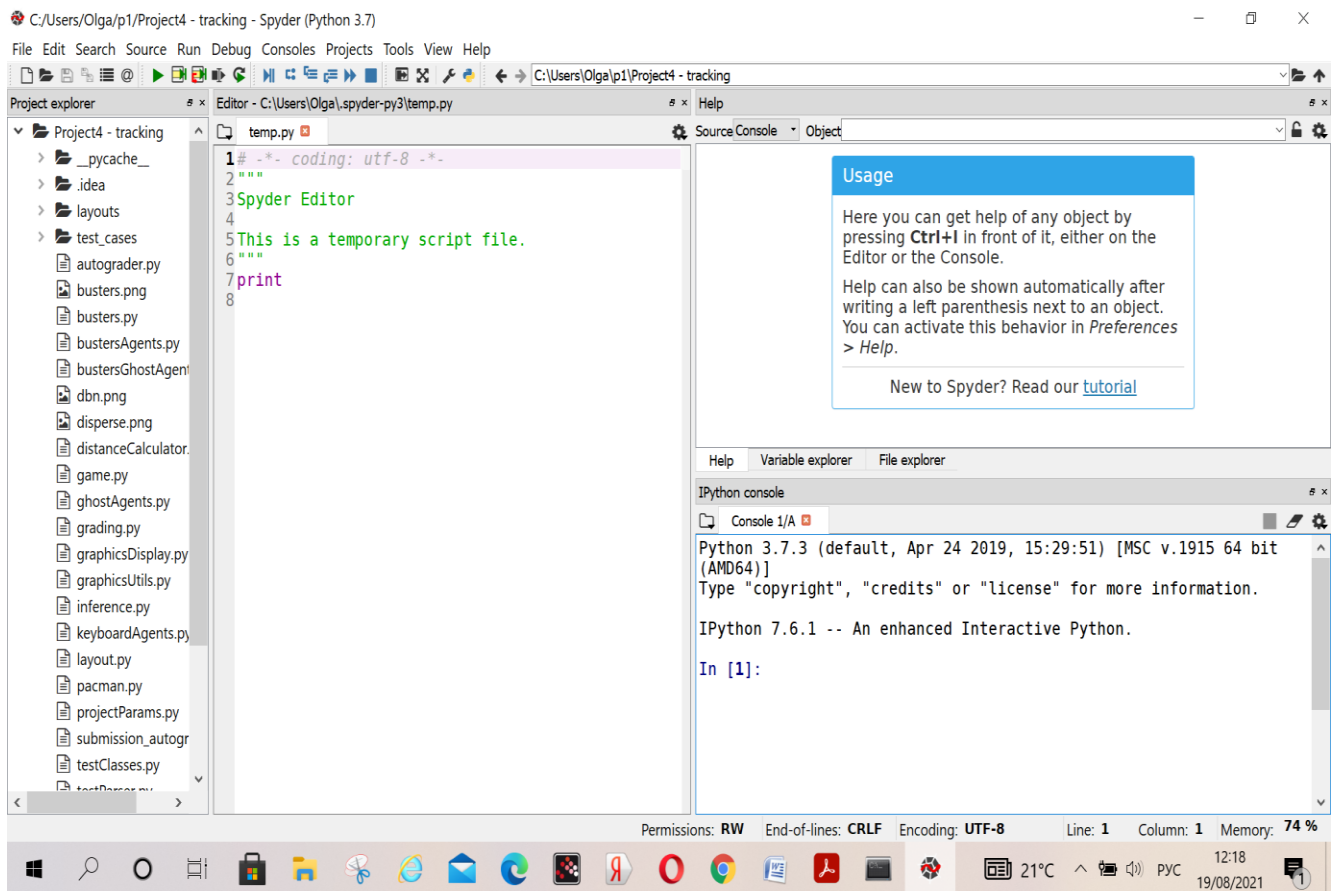


Рисунок 1.2 – Основной экран интегрированной среды Spyder IDE

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	+	$f + 7$	<code>f + 7</code>
Вычитание	–	$p - c$	<code>p - c</code>
Умножение	*	$b \cdot m$	<code>b * m</code>
Возведение в степень	**	x^y	<code>x ** y</code>
Деление	/	x/y , или $\frac{x}{y}$, или $x \div y$	<code>x / y</code>
Целочисленное деление	//	$[x/y]$, или $\left\lfloor \frac{x}{y} \right\rfloor$, или $[x \div y]$	<code>x // y</code>
Остаток от деления	%	$r \bmod s$	<code>r % s</code>

В Python также существуют логические операторы и операции отношений, выполняющие 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`. Строковая константа заключается в одинарные или двойные кавычки: `'hello'`, `"hello"`. Оператор `“+”` позволяет выполнить конкатенацию строк:

```

>>>'искусственный'+”интеллект”
'искусственныйинтеллект'

```

Существует много встроенных полезных 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')
'heellello worelld 11'
```

#замена символов

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, \l, \', \"):

```
>>> 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 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, 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 строковым методом `format`. Собственно, функциональность форматных строк основана на средствах метода `format`. Метод `format` вызывается 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__', '__getattr__', '__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',
'rstrip', '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-й 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_tuple[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 tuple 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, что объекты множества неупорядочены; вы не можете предполагать, что их обход или порядок печати будут одинаковыми на разных машинах!

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 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 может 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]
```



```
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
```

1.2.8. 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))`. Лям-

бда-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
```

Вспользуемся 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]
```

1.2.9. Классы и объекты Python

Класс инкапсулирует данные и функции для взаимодействия с этими данными. Пример определения класса `FruitShop` (магазин фруктов):

```
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__`, похоже, 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`.

1.2.10. Дополнительные советы и хитрости Python

Рассмотрим несколько полезных советов.

1. Используйте 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 `ru`

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 “too many values to unpack error”:

```
(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))
```

Проблема: `AttributeError`: объект «`list`» не имеет атрибута «`length`» (или чего-то подобного)

Решение: определение длины списков выполняется с помощью `len(имя списка)`.

Проблема: изменения в файле не вступили в силу.

Решение: Убедитесь, что вы сохраняете все свои файлы после любых изменений. Если вы редактируете файл в окне, отличном от того, которое вы используете для выполнения, убедитесь, что вы выполнили перезагрузку `reload(модуля)`, чтобы гарантировать, что ваши изменения будут отражены, `reload` работает аналогично `import`.

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 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 автооценителя
 testClasses.py: код автооценителя
 codingDiagnosticTestClasses.py: тестовые классы для этого лабораторного задания (проекта)
 projectParams.py: параметры

Команда python autograder.py оценит ваше решение для всех трех задач (вопросов). Если запустить автооценитель 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 пройдены. 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 автооценщик 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 из кортежей `(fruit, pound)` и возвращает стоимость заказа. Если в списке 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 `[('яблоки', 2,0), ('груши', 3,0), ('лаймы', 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 магазин `FruitShop`, где ваш заказ будет иметь наименьшую стоимость. Пожалуйста, не меняйте имя файла или имена переменных. Обратите внимание, что реализация класса `shop` предоставляется вам в файле `shop.py`. Запускайте `python autograder.py`, пока не будут пройдены все тесты, и вы не получите итоговые оценки.

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 Spider 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, сегментов 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?

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 в 1980 году.

В обучающей среде предусмотрены четыре темы заданий: поиск в пространстве состояний, многоагентный поиск, вероятностный вывод и обучение с подкреплением. Каждый тема требует от студентов изучения универсальных алгоритмов ИИ и их практического внедрения в обучающую среду.

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 есть призраков, зарабатывая очки.

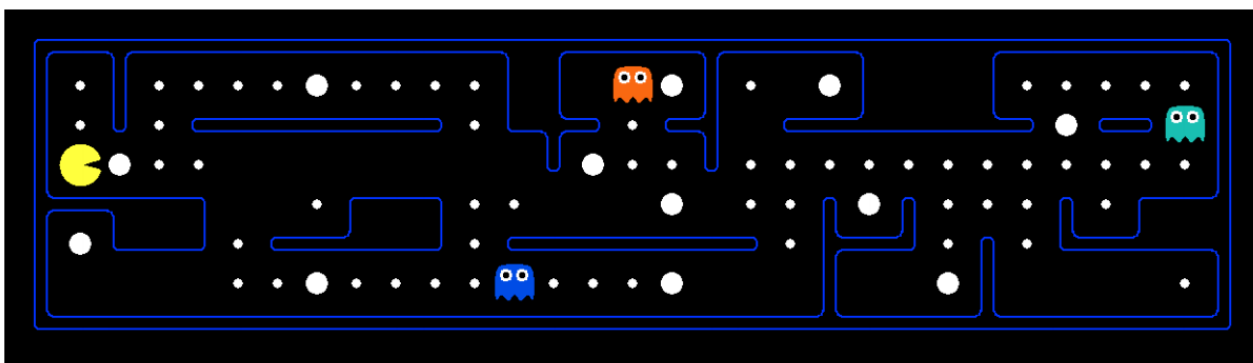


Рисунок 2.1 – Экран обучающей среды AI Pacman

2.2.2. Агенты

Центральной проблемой искусственного интеллекта является создание **рационального агента** – сущности, которая имеет цели или предпочтения 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 интеллектуальную функцию человека – 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 (его конфигурацию в заданной среде) и то, как агент может попасть 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

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

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. $f_n \circ f_{n-1}(\dots(f_2(f_1(S_0)))) \in G$. 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):

- **пространства состояний (state space)** — множество всех состояний, которые возможны мире агента;
- **начального состояния (start state)** — состояния, в котором агент существует изначально;
- **функции-преемника (successor function)** — функции, которая принимает на входе состояние и действие и вычисляет стоимость выполнения этого дейст-

вия, а также находит состояние-преемник (**successor**), в котором находился бы мир, если бы выполнил это действие.

- **целевого теста** — **функции (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-преемников** к некоторой вершине графа с целью получения всех ее дочерних вершин (преемников) называется **раскрытием вершины**.

2.2.4. Формулировки задач поиска в среде AI Pacman

Рассмотрим вариант игры, при котором в лабиринте находится только Pacman и пищевые гранулы. В этом случае можно сформулировать две отдельные задачи поиска: нахождение пути в лабиринте и поедание всех точек. В первом случае мы будем пытаться решить проблему оптимального перехода из позиции $(x_1; y_1)$ в позицию $(x_2; y_2)$ 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: - <i>состояния</i>: координаты (x, y) местоположений; - <i>действия</i>: Север, Юг, Восток, Запад; - <i>преемник</i>: обновить только местоположение; - <i>проверка цели</i>: $(x, y) = \text{END?}$	Поедание всех точек (гранул): - <i>состояния</i>: координаты (x, y) местоположений, логические значения точек (true/false); - <i>действия</i>: Север, Юг, Восток, Запад; - <i>преемник</i>: обновить местоположение и логические значения точек; - <i>проверка цели</i>: все ли логические значения точек ложны?
--	--

Обратите внимание, что для задачи нахождения пути состояния содержат меньше информации, чем состояния для задачи поедания всех точек. 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, пройденном 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 (стратегии) 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 следующих критериев, указанных в [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 состояния 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 поиске на дереве 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  
        дочерней 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 $B+B^2+B^3+\dots+B^d+(B^{d+1}-B)$ [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 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 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 из списка **OPEN** выбирать для раскрытия вершину с наименьшей стоимостью, то рассмотренная выше процедура будет обеспечивать нахождение **оптимального** решения по критерию стоимости (при условии модификации стоимости вершины в списке **OPEN**, при обнаружении пути с меньшей стоимостью). Стоимость раскрытия $g(V_i)$ некоторой вершины V_i равна

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

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 равны $O(B^{1+C/c})$, где C — стоимость оптимального решения, c — минимальная стоимость каждого 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 этапов небольшой стоимости, прежде чем перейти к исследованию путей, в которые входят крупные, но возможно более полезные этапы [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 [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** равна $O(B^m)$, где m — максимальная глубина дерева поиска (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 самого поверхностного решения. Конечно, здесь допускается повторное формирование 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 $O(Bd)$).

Так как при поиске в ширину некоторые вершины формируются на глубине $d+1$, то поиск с итеративным углублением фактически осуществляется быстрее, чем поиск в ширину, несмотря на повторное формирование состояний.

2.2.7. Общие сведения о программировании в среде AI Pacman

Код среды AI Pacman, используемый в данной лабораторной работе, состоит из нескольких файлов на языке Python. 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_лаб_2_3.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 поддерживаемых типов, таких как AgentState, Agent, Direction и Grid.
util.py	Полезные структуры данных для реализации алгоритмов поиска.

Поддерживаемые 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	Автооценщик
testParser.py	Парсер тестов автооценщика и файлы решений
testClasses.py	Общие классы автооценщика
test_cases/	Папка, содержащая тесты для каждого из заданий (вопросов)
searchTestClasses.py	Специальные тестовые классы автооценщика 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_лаб_2_3.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 "I"):

```
python pacman.py
```

Pacman живет в мире разветвленных коридоров и угощений, представленных пищевыми гранулами. Эффективная навигация в этом мире будет первым шагом Pacman в освоении своей 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, набрав **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 (30 баллов). Поиск в глубину

Будем использовать поискового агента **SearchAgent**, реализованного в **searchAgents.py**, который планирует путь в мире Pacman, а затем выполняет его пошагово. Ваша задача состоит в том, чтобы реализовать поисковые алгоритмы для составления плана поиска.

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 автооценщиком.

Подсказка: все алгоритмы поиска очень похожи. Алгоритмы для DFS, BFS, UCS отличаются только деталями управления списком **OPEN**. Сконцентрируй-

тесь 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. Написанный код 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
```

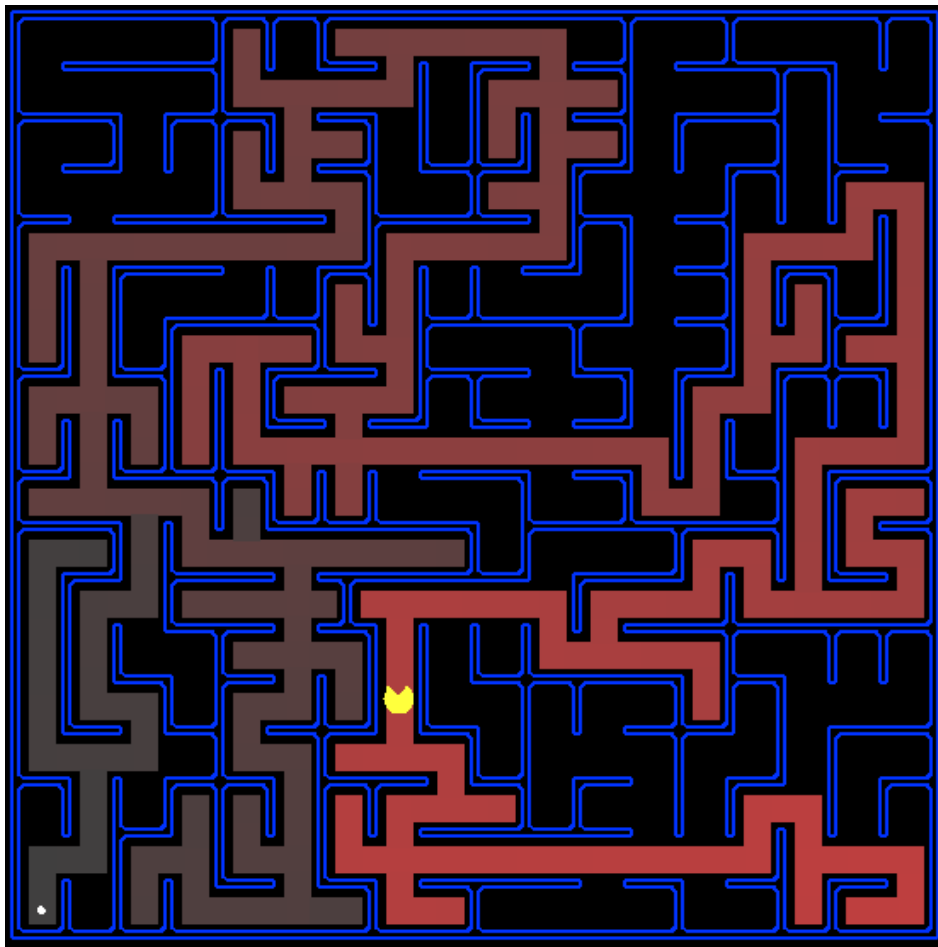


Рисунок 2.2 – Схема большого лабиринта (bigMaze)

Схема лабиринта Pacman (рисунок 2.2) показывает наложение исследованных состояний и порядок, в котором эти состояния были исследованы (более яркий красный означает более раннее исследование). Ответьте на два вопроса: Вы ожидали такой порядок исследования? На самом ли деле Pacman проходит все исследованные 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 наименее затратное решение? Если нет, подумайте, что не так с поиском в глубину.

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

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

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

Задание 2 (30 баллов). Поиск в ширину

Реализуйте алгоритм поиска в ширину (BFS) в функции **breadthFirstSearch** в файле **search.py**. Напишите алгоритм поиска пути на графе, который избегает раскрытия уже посещенных состояний. 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 Распан движется слишком медленно, то попробуйте опцию **--frameTime 0**.

Примечание. Если вы написали свой поисковый код в достаточно обобщенной форме, то он должен также хорошо работать для задачи поиска игры в восемь без каких-либо изменений.

```
python eightpuzzle.py
```

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

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

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

Задание 3 (40 баллов). Поиск на основе алгоритма равных цен

Хотя BFS найдет путь к цели с наименьшим количеством действий, можно попытаться найти пути, которые являются «лучшими» в других смыслах. 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 автооценителя:

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

Если возникают ошибки, то исправьте их. После успешного прохождения тестов автооценителя 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.

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.

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 2.2.2. 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 дистрибутива **МИСИИ_лаб_2_3.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 Расман (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 Расман, 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.

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 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 функции**, 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 A^* - 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 A -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**, сравнить 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 A -алгоритма

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

- 1) A -алгоритм соответствует алгоритму равных цен, если $h(n)=0$;
- 2) A -алгоритм **гарантирует оптимальное решение**, если $\hat{h}(n) \leq h(n)$; в этом случае он называется **A^* – алгоритмом**. A^* -алгоритм недооценивает за-

траты 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** $\hat{h}(n_i) \leq \hat{h}(n_j) + c(n_i, n_j)$, 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^* эвристически более сильный, 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) A^* -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 A^* -алгоритма может снижаться из-за того, что вершина, находящаяся в списке CLOSED, может попадать обратно в список OPEN и затем повторно раскрываться.

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

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

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 A^* -алгоритм проверяет все узлы в контуре меньшей стоимости, прежде чем перейдет к проверке узлов следующего контура.

A^* -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 длины решения. По

этой причине на практике стремление находить оптимальное решение не всегда оправдано. Иногда вместо этого целесообразно использовать варианты А-поиска, позволяющие быстро находить неоптимальные решения, или разрабатывать более точные эвристические функции, но не строго допустимые.

Так как при А*-поиске хранятся все раскрытые узлы, фактические ресурсы пространства исчерпаются гораздо раньше, чем временные ресурсы. С этой целью применяют *А*-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 **МА*** [4].

3.3. Задания для выполнения

Задание 1 (20 баллов). А*- поиск

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

Протестируйте свою реализацию А*-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=manhattanHeuristic
```

Вы должны увидеть, что А*- 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 А*-поиска все тестовые примеры автооценителя:

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

Задание 2 (20 баллов). Поиск всех углов

Настоящая сила А*-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 не-

которых лабиринтов, таких как **tinyCorners**, кратчайший путь не всегда ведет к ближайшей точке в первую очередь!

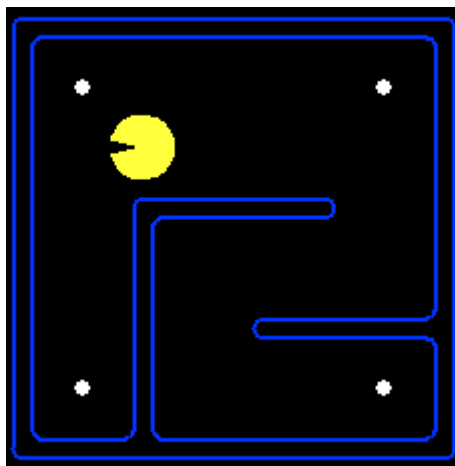


Рисунок 2.3. – Задача поиска углов

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 приемников 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 автооценителя:

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

Задание 3 (20 баллов). 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 автооценщика. Вам нужна нетривиальная эвристика, которая сокращает общее время вычислений, хотя для этого задания автооценщик будет проверять только количество узлов (помимо обеспечения разумного ограничения по времени).

Оценивание: ваша эвристика должна быть нетривиальной неотрицательной 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	0/20
максимум 2000	7/20
максимум 1600	15/20
максимум 1200	20/20

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 автооценителя

```
python autograder.py -q q6
```

Задание 4 (20 баллов). 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 при простом лабиринте **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 5 баллов. 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	5/20
максимум 15000	10/20
максимум 12000	15/20
максимум 9000	20/20
максимум 7000	25/20 (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 автооценителя:

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

Задание 5 (20 баллов). 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 пути в этом лабиринте субоптимально, менее чем за секунду 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 автооценителя:

```
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_лаб_2_3.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
    xy2 = problem.goal
    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 реализовать в конструкторе класса **CornersProblem** возможность хранения значения атрибута **self.startingGameState**.

Метод **getStartState** должен вернуть начальное состояние, которое включает в себя **self.startingPosition**, и пустой кортеж **()**. 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 комментарий внутри цикла **for** по возможным действиям (направлениям перемещения) агента:

```
# Добавьте состояние-приемник в список приемников, если действие является
# допустимым
# Ниже фрагмент кода, который проверяет, 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, 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** соответствуют 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(corners[0] - position[0]) + abs(corners[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 др. (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 допусти-

мую 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 автооценителя, 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 авто-

оценивания, 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 А-алгоритмом.

3.6.4. Сформулируйте алгоритм подъема в гору.

3.6.5. Сформулируйте алгоритм глобального выбора первой наилучшей вершины.

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

3.6.7. Сформулируйте и объясните свойства А-алгоритма.

3.6.8. Какой А-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.

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 пытается максимизировать, а его противник пытается минимизировать. Например, в игре Распап такая переменная соответствует набранному баллам, которые Распап пытается максимизировать, поедая гранулы, в то время как призрак пытается 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: статические и динамические [1]. **Статические оценки** приписываются терминальным вершинам (состояниям) дерева игры. Вычисление этих оценок реализуются в виде **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*.

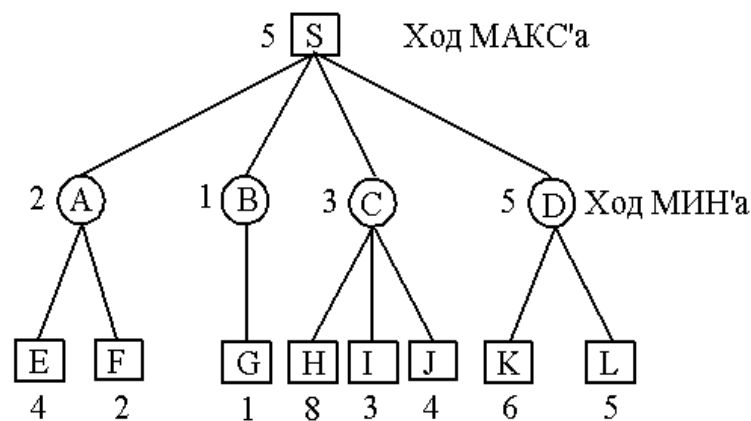


Рисунок 4.1 – Распространение оценок при минимаксной игре

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) 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 =  $-\infty$ 
    for succ in successor(state): # для всех дочерних состояний
        v=max(v, value(succ)) # рекурсивный вызов value
    return v

def min_value(state):
    v =  $+\infty$ 
    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 А, В и С (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 Х. В этом состоянии ход выбирает игрок С, анализируя векторы оценок дочерних состояний: (1, 2, 6) и (4, 2, 3). Поскольку для игрока С состояние с $V_C = 6$ предпочтительнее, то игрок С в состоянии Х выбирает первый ход и состоянию Х приписывается вектор оценок (1, 2, 6).

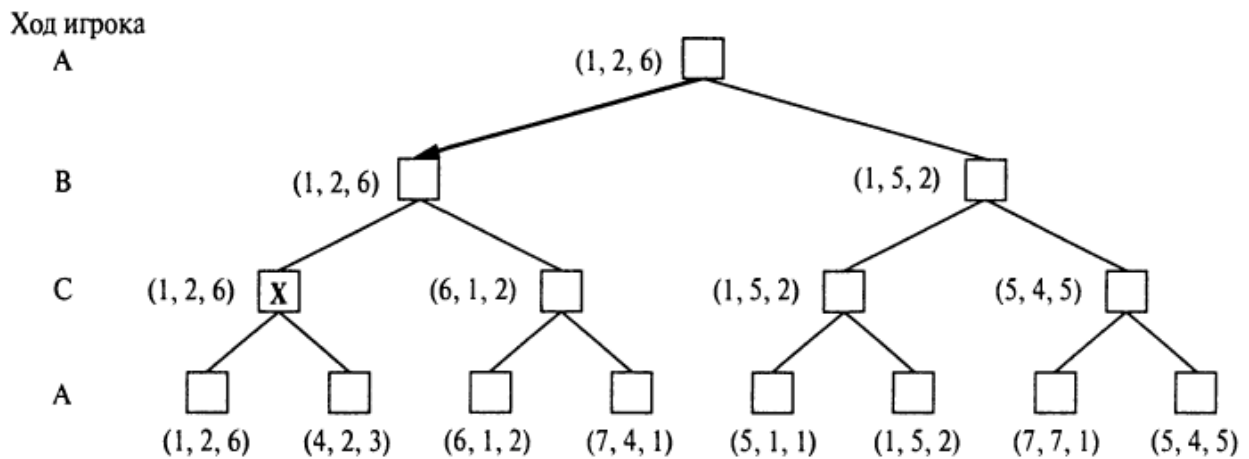


Рисунок 4.2 – Дерево игры с несколькими игроками

Действуя аналогично при всех других состояниях игры, получаем для игрока А в корневом состоянии вектор оценок, равный (1, 2, 6), т.е. игрок А 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 отсека некоторых поддеревьев поиска. Предполагается, что поиск осуществляется сверху вниз и слева направо.

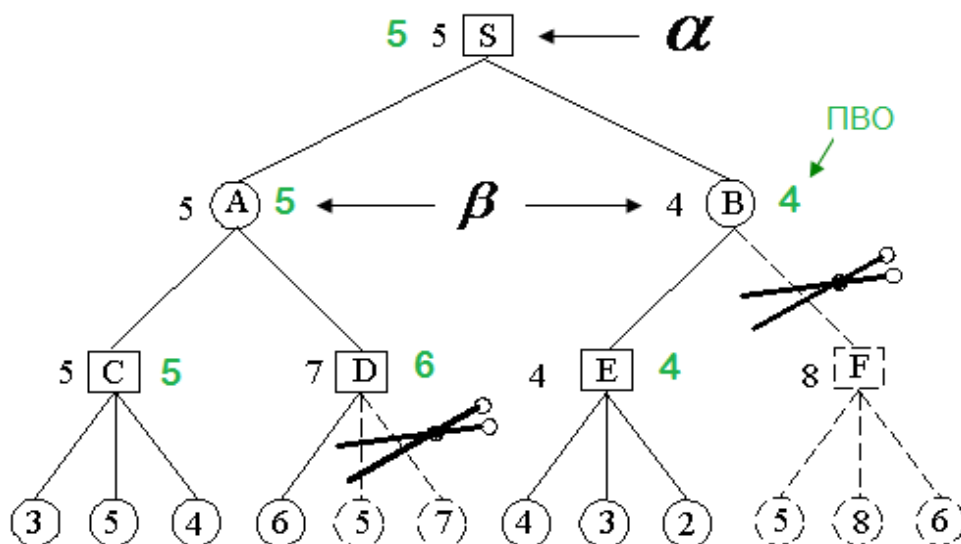


Рисунок 4.3 – Альфа-бета поиск

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,  $\alpha$ ,  $\beta$ ):
    v =  $-\infty$ 
    for succ in successor(state):
        v = max(v, value(succ,  $\alpha$ ,  $\beta$ ))
        if v  $\geq$   $\beta$ : return v          # альфа отсечение
         $\alpha$  = max( $\alpha$ , v)
    return v

def min_value(state,  $\alpha$ ,  $\beta$ ):
    v =  $+\infty$ 
    for succ in successor(state):
        v = min(v, value(succ,  $\alpha$ ,  $\beta$ ))
        if v  $\leq$   $\alpha$ : return v        # бета отсечение
         $\beta$  = min( $\beta$ , 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 функций $f_i(s)$:

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

где каждая функция $f_i(s)$ вычисляет некоторую характеристику 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 королей у агента, количество пешек у противника и количество королей 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:

$$V(s) = \sum_i p(s_i | s) V(s_i), \quad (4.1)$$

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 **Exprectimax** пропорциональна $O(b^m n^m)$, где n — количество вариантов выпадения жребия.

4.3. Задания для выполнения

Задание 1 (16 баллов). 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
```

Для приведения оценки, выставленной автооценщиком, к 100-балльной итоговой шкале её необходимо умножить на 4. Не тратьте слишком много времени на усовершенствование решения этого задания, поскольку основные задания лабораторной работы впереди.

Задание 2 (5 баллов). 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 Расман и последовательным действиям всех агентов-призраков.

Автооценщик определит, исследует ли ваш агент правильное количество игровых состояний. Это единственный надежный способ обнаружить некоторые очень тонкие ошибки в реализациях минимакса. В результате автооценщик будет очень требователен к числу вызовов метода **GameState.generateSuccessor**. Если метод будет вызываться больше или меньше необходимого количества раз, автооценщик 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 Расман. 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 Расман 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 рефлексным агентом. Планирующие агенты оценивают будущие состояния, тогда как рефлексные 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
```

- Расман 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 зна-

ет, 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 Расман в этом случае нападает на ближайший призрак.

Задание 3 (5 баллов). 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 скоростью 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.generateSuccessor** 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 на игре расман. Чтобы запустить код без графики, используйте команду:

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

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

Задание 4 (5 баллов). 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 отладить 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 Расман.

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 Расман, 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 Exрестімах приведет к тому, что Распан будет проигрывать некоторые тесты. Это не создаст проблем при автооценивании. Ваш агент пройдет тесты.

Задание 5 (6 баллов). Функция оценки

Реализуйте лучшую оценочную функцию для игры Распан 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, Распан 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 автооценителя, 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 автооценителем 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 Exрестімах.

4.4.2. Использовать для выполнения лабораторной работы файлы из архива **МИСИИ_лаб_4.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 Распан, включая призраков. При этом потребуется реализо-

вать как минимаксный поиск, так и `exrestimax` поиск, а также разработать оценочные функции.

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

```
python autograder.py
```

Для проверки решения конкретного задания, например 2-го, автооценщик 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 автооценщик 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:	
<code>multiAgents.py</code>	Здесь будут размещаться все ваши мультиагентные поисковые агенты, которых вы определите.
Файлы, которые необходимо просмотреть	
<code>pacman.py</code>	Основной файл, из которого запускают Pacman. Этот файл описывает тип <code>Pacman GameState</code> , который используется в лабораторных работах.
<code>game.py</code>	Логика, лежащая в основе мира Pacman. Этот файл описывает несколько поддерживаемых типов, таких как <code>AgentState</code> , <code>Agent</code> , <code>Direction</code> и <code>Grid</code> .
<code>util.py</code>	Полезные структуры данных для реализации алгоритмов поиска.
Поддерживающие файлы, которые можно игнорировать:	
<code>graphicsDisplay.py</code>	Графика Pacman
<code>graphicsUtils.py</code>	Графические утилиты
<code>textDisplay.py</code>	ASCII графика Pacman
<code>ghostAgents.py</code>	Агенты, управляющие привидениями
<code>keyboardAgents.py</code>	Интерфейс клавиатуры для управления игрой
<code>layout.py</code>	Код для чтения файлов схем и хранения их содержимого
<code>autograder.py</code>	Автооценщик
<code>testParser.py</code>	Парсер тестов автооценщика и файлы решений
<code>testClasses.py</code>	Общие классы автооценщика
<code>test_cases/</code>	Папка, содержащая тесты для каждого из заданий (вопросов)
<code>multiagentTestClasses.py</code>	Специальные тестовые классы автооценщика 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 псевдокод 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. $p(s_i | s) = 1/N$, 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.

ЛАБОРАТОРНАЯ РАБОТА № 5

ИССЛЕДОВАНИЕ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ

5.1. Цель работы

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

5.2. Краткие теоретические сведения

5.2.1. Недетерминированный поиск

Ранее были рассмотрены представления интеллектуальных задач в пространстве состояний и соответствующие методы поиска решений, а также способы представления задач в условиях противодействия и состязательные методы поиска решений.

Рассмотрим методы поиска в условиях, когда действие агента, выполненное в некотором состоянии, может приводить ко многим возможным новым состояниям с определенной долей вероятности. Такие задачи поиска, в которых поведение агента характеризуется степенью неопределенности, относятся к **недетерминированным задачам поиска**. Они могут быть решены с помощью моделей, известных как **марковские процессы принятия решений (Markov Decision Processes – MDP)**.

5.2.2. Марковские процессы принятия решений

Марковский процесс принятия решений (MDP) 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

$$\text{MDP} = \{ S, A, T, R, \gamma \}.$$

При определении MDP также указывают **начальное 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 процесса зависит вероятностно только от текущего состояния, то есть $P(s'|s, a)$. Это подобно поиску, в котором функция-приемник использует только предшествующее состояние (а не историю состояний).

Решение задачи, представляемой в виде MDP, сводится к поиску **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 в дискретные моменты времени 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 по всем временным шагам, которое можно представить в виде **функции полезности**

$$U([s_0, a_0, s_1, a_1, s_2, \dots]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + R(s_2, a_2, s_3) + \dots \quad (5.1)$$

Если число шагов конечное, то это **эпизодическая задача** и награды суммируются по эпизоду (до терминального состояния). Сумма наград, полученных агентом, также называется **возвратом**. Сокращенно это можно записать в виде

$$U([r_0, r_1, r_2, \dots]) = r_0 + r_1 + r_2 + \dots; \quad (5.2)$$

где $r_0, r_1, \dots$ – 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 (возврат) с **коэффициентом дисконтирования** в следующем виде:

$$U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \quad (5.3)$$

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 ($\gamma<1$):

$$U([r_0, \dots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1 - \gamma) \quad (5.4)$$

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 :

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right], \quad (5.5)$$

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 :

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

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 так:

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi \right]. \quad (5.7)$$

5.2.3. MDP дерево поиска и уравнение Беллмана

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

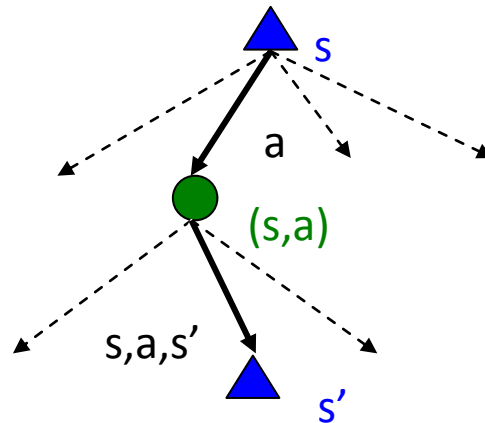


Рисунок 5.1. – MDP дерево поиска

Для решения задачи MDP используют уравнение Беллмана. Когда мы ищем решение задачи, представленной в виде MDP, на самом деле нас интересует нахождение оптимальных функций ценности и политики.

Оптимальной функцией ценности $V^*(s)$ называется функция ценности, которая обеспечивает максимальную ценность при следовании определенной политике [7]:

$$V^*(s) = \max_{\pi} V^{\pi}(s). \quad (5.8)$$

Оптимальную функцию ценности-состояния удобно вычислять как максимум Q-функции по действиям a :

$$V^*(s) = \max_a Q^*(s, a). \quad (5.9)$$

Ценность q-состояния (s,a) можно определить выражением (см. рисунок 5.1):

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] . \quad (5.10)$$

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

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

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

5.2.4. Итерации по значениям ценности состояний

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

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

Алгоритм предусматривает следующие шаги:

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

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

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

5.2.5. 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:

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

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

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

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

5.2.6. 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:

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

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

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

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

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

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

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

Важно отметить, что политика в этом случае сходится к оптимальной политике. Кроме этого, сходжение политики происходит быстрее сходжения значений ценности состояний.

5.2.7. Обучение с подкреплением

Обучение с подкреплением (RL, reinforcement learning) — область машинного обучения, в которой обучение осуществляется посредством взаимодействия агента с окружающей средой (рисунок 5.2) [7, 8]. В среде RL вы не указываете агенту, что и как он должен делать, вместо этого **вы даете агенту награду за каждое выполненное действие**. Тогда агент начинает выполнять действия, которые максимизируют вознаграждение.

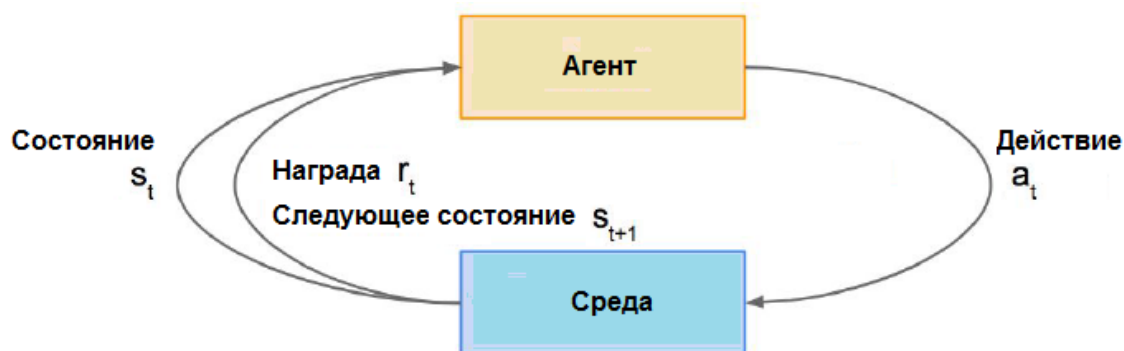


Рисунок 5.2 – Обучение с подкреплением

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 RL, прежде, чем начнет **эксплуатировать (exploitation)** полученную политику.

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

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

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

5.2.8. 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')).

5.2.9. 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.

5.2.10. 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 никаких знаний T, R ;
- вычисляет средние значения ценностей, используя просто выборки переходов.

Основным недостатком алгоритма являются значительные временные затраты.

5.2.11. 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 $V(s)$:

- выборка $V(s)$:

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

- скользящее среднее выборок:

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

или

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

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

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

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

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

5.2.12. Q-обучение

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

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

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

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

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

и их скользящем усреднении

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

или

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

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

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

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

5.2.14. Q-обучение с 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 отображают состояния на действительные числа и которые фиксируют важные свойства состояний. Примеры признаков состояний для игры Пакман:

- расстояние до ближайшего призрака;
- расстояние до ближайшей гранулы;
- число призраков;
- $1/(\text{расстояние до призрака})^2$;
- находится ли Пакман в ловушке? (0/1);
- и др..

Используя вектор признаков, мы можем аппроксимировать функцию ценности q -состояния, например, в виде линейной комбинации признаков с весами:

$$Q(s, a) = w_1 \cdot f_1(s, a) + w_2 \cdot f_2(s, a) + \dots + w_n \cdot f_n(s, a) = \vec{w} \cdot \vec{f}(s, a) \quad (5.25)$$

где $\vec{f}(s, a) = [f_1(s, a) \ f_2(s, a) \ \dots \ f_n(s, a)]^T$ – вектор признаков q -состояния (s, a) , а $\vec{w} = [w_1 \ w_2 \ \dots \ w_n]$ – 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*

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s, a) . \quad (5.26)$$

и вычислить новые значения ценности $Q(s, a)$ в соответствии с (5.25);

- 4) повторять шаги 1)-3) до схождения ценности q -состояний.

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.

Если в качестве аппроксиматора $Q(s, a)$ используется глубокая нейронная сеть, то говорят о **глубоком q -обучении**. Сеть, выполняющую указанную аппроксимацию, называют **q -сетью** [8]. Соответственно, правило обновления весов нейросети в этом случае будет отличаться от (5.26).

5.2.15. Функции разведки (исследования)

При использовании рассмотренного алгоритма q -обучения с аппроксимацией требуется вручную управлять коэффициентом ϵ . Этого можно избежать, если применить **функции разведки (исследования)**, которые используют модифицированное правило обновления ценности q -состояний, предоставляющее предпочтения тем состояниям, которые посещаются реже:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \cdot [R(s, a, s') + \gamma \max_{a'} f(s', a')] , \quad (5.27)$$

где f – функция разведки. Имеется определенная степень свободы при выборе функции разведки, но обычно полагают, что

$$f(s, a) = Q(s, a) + \frac{k}{N(s, a)}, \quad (5.28)$$

где k – некоторая предопределенная константа, $N(s, a)$ – число посещений 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 (5.28) 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.

5.3. Задания для выполнения

Задание 1 (4 балла). Итерации по значениям

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

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

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

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

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

Важно: используйте «пакетную» версию итерации по значениям, где каждый вектор ценности состояний V_k вычисляется на основе предыдущих значений

вектора V_{k-1} , а не на основе «онлайн» версии, где один и тот же вектор обновляется по месту расположения. Это означает, что при обновлении значения состояния на итерации k на основе значений его состояний-предшественников, значения состояний-предшественника, используемые в вычислении обновления, должны быть значениями из итерации $k-1$ (даже если некоторые из состояний-предшественников 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 обрабатывайте случай, когда состояние не имеет доступных действий в MDP (подумайте, что это означает для будущих вознаграждений).

Чтобы протестировать вашу реализацию, запустите автооценщик:

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

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

```
python gridworld.py -a value -i 5
```

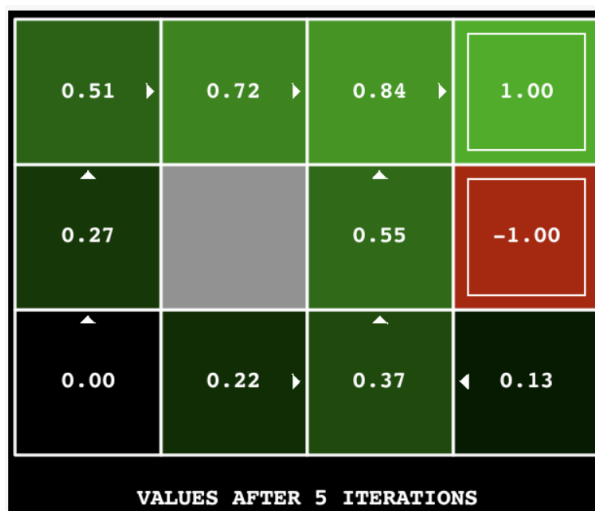


Рисунок 5.3 – Состояние среды BookGrid после 5 итераций

Оценивание: ваш агент, использующий итерации по значениям, будет оцениваться с использованием новой схемы клеточного мира. Будут проверяться ценности состояний, Q -ценности и политики после заданного количества итера-

ций, а также с учетом выполнения условия сходимости (например, после 100 итераций).

Задание 2 (1 балл). Анализ перехода через мост

BridgeGrid - это клеточный мир с высоким вознаграждением в целевом конечном состоянии, к которому ведет узкий «мост», по обе стороны от которого находится пропасть с высоким отрицательным вознаграждением (рисунок 5.4). Агент начинает движение из состояния с низким вознаграждением. С коэффициентом дисконтирования, по умолчанию равным 0,9, и уровнем шума, по умолчанию равным 0.2, оптимальная политика не обеспечивает прохождения моста. Измените только ОДИН из параметров - коэффициент дисконтирования или уровень шума, чтобы при оптимальной политике агент попытался пересечь мост. Поместите свой ответ в код функции **question2()** в файле **analysis.py**. (Шум соответствует тому, как часто агент попадает в незапланированное состояние-преемник при выполнении действия). Значения по умолчанию соответствуют вызову:

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



Рисунок 5.4 – Состояние среды BridgeGrid после 100 итераций

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

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

Задание 3 (5 баллов). Реализация политик

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

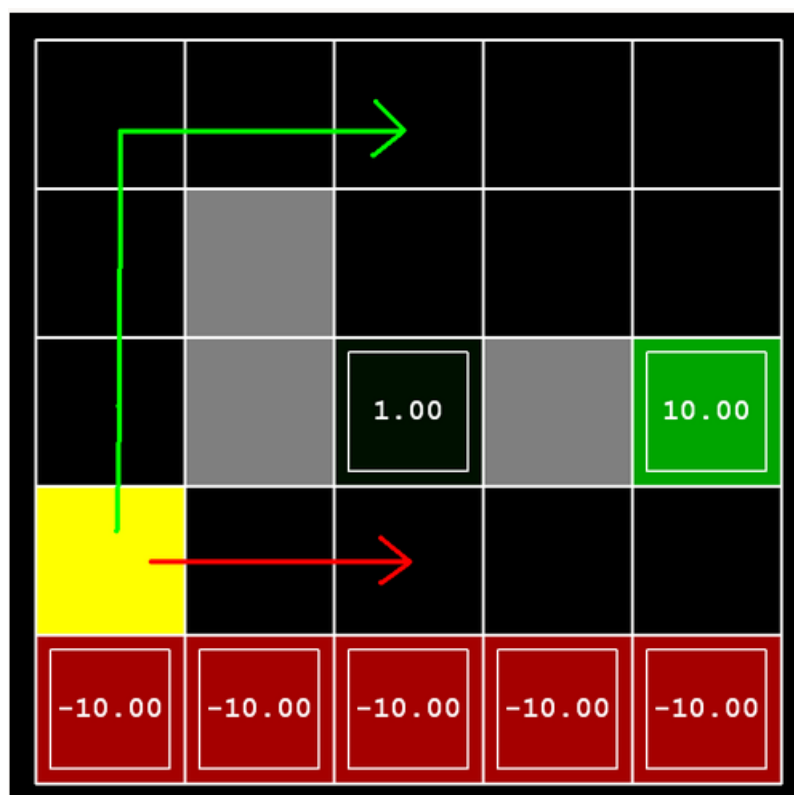


Рисунок 5.5 – Состояния среды DiscountGrid

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

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

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

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

Чтобы проверить свой выбор параметров, запустите автооцениватель:

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

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

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

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

Задание 4 (1 балл). Асинхронная итерация по значениям

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

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

Класс **AsynchronousValueIterationAgent** является наследником от класса **ValueIterationAgent** из задания 1, поэтому единственный метод, который требуется изменить, - это **runValueIteration**. Поскольку конструктор суперкласса вызывает

метод **runValueIteration**, достаточно его переопределения в классе **AsynchronousValueIterationAgent**, чтобы изменить поведение агента.

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

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

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

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

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

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

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

Задание 5 (3 балла). Итерации по приоритетным значениям

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

В этом задании необходимо реализовать упрощенную версию стандартного алгоритма итераций по приоритетным значениям, который описан в [статье](#). Будем использовать адаптированную версию этого алгоритма. Во-первых, определим предшественниками состояния s все состояния, которые имеют ненулевую вероятность достижения s путем выполнения некоторого действия a . Кроме того, введем параметр **theta**, который будет представлять некоторый порог приращения функции ценности (устойчивость к ошибкам) при принятии решения об обновлении значения состояния. Сформулируем алгоритм, который вы должны реализовать:

1. Вычислите предшественников всех состояний;
2. Инициализируйте пустую приоритетную очередь;

3. Для каждого нетерминального состояния **s** выполните: (примечание: чтобы автооценщик работал корректно необходимо перебирать состояния в порядке, возвращаемом **self.mdp.getStates ()**):

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

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

4. Для **iteration in 0, 1, 2, ..., self.iterations - 1** выполнить:

4.1. Если приоритетная очередь пуста, завершите работу;

4.2. Извлеките (метод **pop**) состояние **s** из очереди с приоритетами;

4.3. Обновите значение ценности **s** (если это не конечное состояние) в **self.values**;

4.4. Для каждого **p** предшественника состояния **s** выполните:

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

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

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

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

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

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

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

Оценивание: агент с итерациями приоритетным значениям будет оцениваться с использованием новой схемы клеточного мира. Будут проверены q-

ценности и политики после фиксированного количества итераций и при достижении сходимости (например, после 1000 итераций).

Задание 6 (4 балла). Q-обучение

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

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

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

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

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

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

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

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

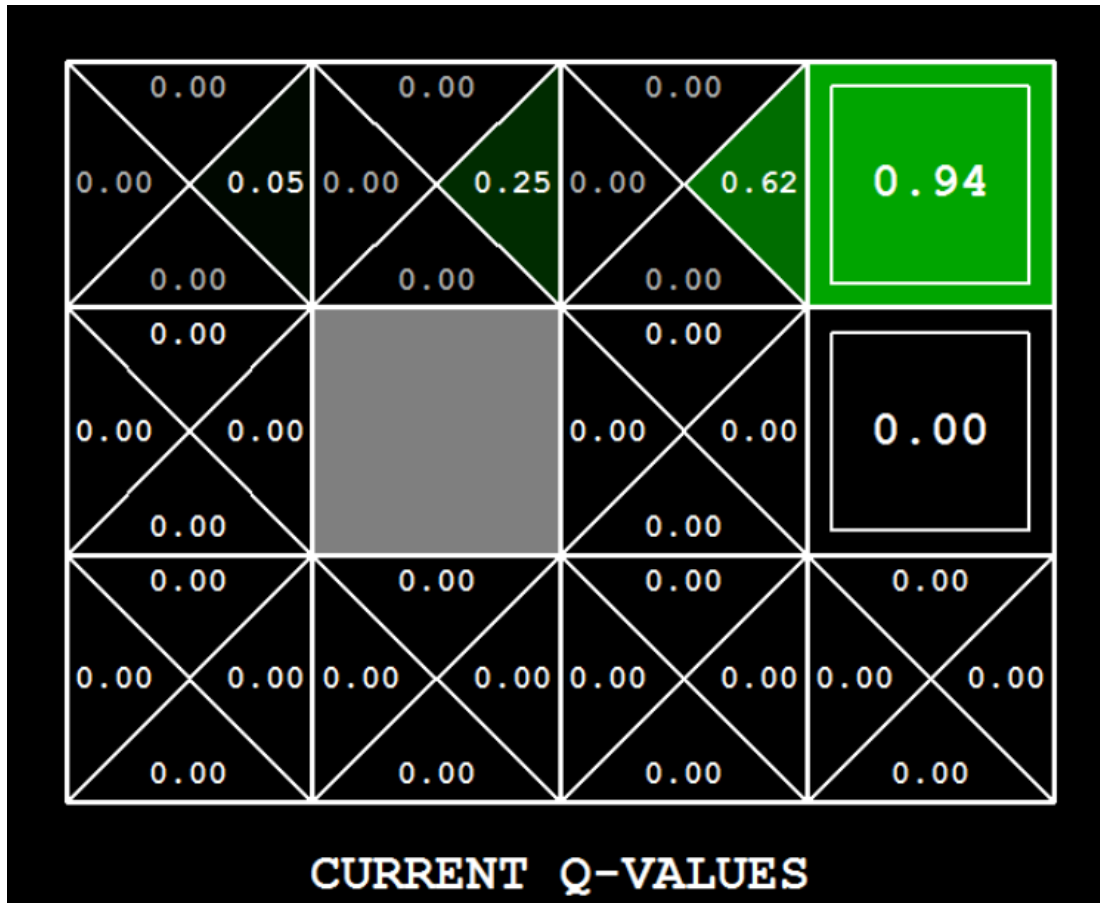


Рисунок 5.6. – Результаты q-обучения

Оценивание: проверяется, обучается ли агент тем же значениям q-ценности и политике, что и наша эталонная реализация на одном и тот же наборе примеров. Чтобы оценить вашу реализацию, запустите автооцениватель:

```
python autograder.py -q q6
```

Задание 7 (2 балла). Эпсилон-жадная стратегия

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

Вы можете выбрать действие из списка случайным образом, вызвав функцию **random.choice**. Вы можете смоделировать двоичную переменную с вероятностью успеха **p**, используя вызов **util.flipCoin(p)**, который возвращает True с вероятностью **p** и False с вероятностью **1-p**.

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

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

Ваши окончательные значения q-ценностей должны соответствовать значениям, получаемым при итерации по значениям, особенно на проторенных путях. Однако среднее вознаграждение будет ниже, чем предсказывают q-ценности из-за случайных действий и начальной фазы обучения.

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

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

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

Чтобы протестировать вашу реализацию, запустите автооценщик:

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

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

```
python crawler.py
```

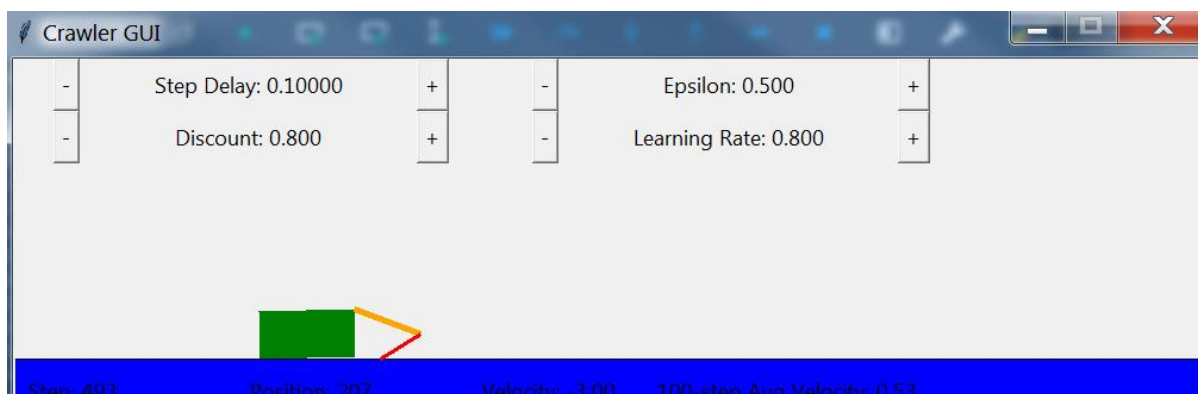


Рисунок 5.7. – Робот Crawler

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

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

Задание 8 (1 балл). Переход по мосту

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

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

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

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

Чтобы оценить свой ответ, запустите автооценщик:

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

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

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

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

Обратите внимание, что **PacmanQAgent** уже определен для вас в терминах уже написанного агента **QLearningAgent**. **PacmanQAgent** отличается только тем, что он имеет параметры обучения по умолчанию, которые более эффективны для игры Пакман (**epsilon=0.05, alpha=0.2, gamma=0.8**). Вы получите максимальную

оценку за выполнение этого задания, если приведенная выше команда будет выполняться без исключений и агент выигрывает не менее, чем в 80% случаев. Автооценщик проведет 100 тестовых игр после 2000 тренировочных игр.

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

Чтобы оценить задание 9, выполните:

```
python autograder.py -q q9
```

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

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

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

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

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

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

Однако вы обнаружите, что обучение того же агента, казалось бы, на простой среде **mediumGrid** не работает. В нашей реализации среднее вознаграждение за обучение Пакмана остается отрицательным на протяжении всего обучения. Во

время теста он играет плохо, вероятно, проигрывая все свои тестовые партии. Тренировки тоже займут много времени, несмотря на свою неэффективность.

Пакман не может победить на больших игровых полях, потому что каждая конфигурация имеет отдельные состояния с отдельными значениями q -ценности. У Пакмана нет возможности обобщить случаи столкновения с призраком и понять, что это плохо для всех позиций. Очевидно, такой вариант q -обучения не масштабирует большое число состояний.

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

Реализуйте q -обучение с аппроксимацией, которое обеспечивает обучение весов признаков состояний. Дополните описание класса **ApproximateQAgent** в **qlearningAgents.py**, который является подклассом **PacmanQAgent**.

Q -обучение с аппроксимацией предполагает существование признаков функции $f(s, a)$ от пары состояние-действие, которая возвращает вектор $f_1(s, a) \dots f_i(s, a) \dots f_n(s, a)$ из значений признаков. Вам предоставляются для этого возможности модуля **featureExtractors.py**. Векторы признаков - это объекты **util.Counter** (подобны словарю), содержащие ненулевые пары признаков и значений; все пропущенные признаки будут иметь нулевые значения.

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

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

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

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

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

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

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

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

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

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

5.4. Порядок выполнения лабораторной работы

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

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

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

В состав файлов лабораторной работы входит автооцениватель, с помощью которого вы можете оценивать свои решения. 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.
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 в виде 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	Автооцениватель 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 автооценивателя

5.4.4. 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, что когда вы нажимаете клавишу «вверх», агент фактически перемещается на Север только в 80% случаев. Это свойство агента в среде **Gridworld**. Вы можете контролировать многие опции среды **Gridworld**. Полный список опций можно получить по команде:

python gridworld.py -h

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

python gridworld.py -g MazeGrid

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

Примечание: среда **Gridworld** такова, что вы сначала должны войти в пред-терминальное 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 *Rastman*, позиции представлены декартовыми координатами (**x, y**), а любые массивы индексируются [**x**] [**y**], где «север» является направлением увеличения **y** и т. д. По умолчанию большинство переходов получают нулевую награду. Это можно изменить с помощью опции **-r**, которая управляет текущей наградой.

5.4.5. В задании 1 требуется реализовать следующие методы класса **ValueIterationAgent(ValueEstimationAgent)**:

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

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

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

```
Qvalue+=prob*(self.mdp.getReward(state, action, nextstate) +  
self.discount*self.getValue(nextstate)),
```

где **nextstate** и **prob** получают путем операции **in** для множества **self.mdp.getTransitionStatesAndProbs(state, action)**.

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

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

5.4.6. В задании 3 необходимо подобрать значения коэффициента дисконтирования (**answerDiscount**), уровня шума (**answerNoise**) и текущей награды (**answerLivingReward**) для среды **DiscountGrid** (рисунок 5.4). Помните, что с помощью значений текущей награды можно управлять степенью риска: большие положительные награды заставляют агента избегать выхода, умеренные положительные награды исключают риск, умеренные отрицательные допускают риск. Снижение уровня шума понижает вероятность отклонения от выбранного направления движения. Коэффициент дисконтирования определяет важность ближайших наград по отношению к будущим наградам.

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

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

5.4.8. В задании 5 необходимо реализовать алгоритм итераций по приоритетным значениям. При реализации алгоритма следуйте указаниям, сформулированным в самом задании.

Поиск с итерациями по приоритетным значениям фокусируется на обновлении значений состояний, которые более вероятно изменяют политику.

Идея заключается в обновлении тех состояний, для которых изменения более ожидаемы. Если разность

$$|V_{i+1}(s) - V_i(s)|$$

большая, то следует в приоритетном порядке обновить ценность предшествующих состояний s . Соответственно для этого формируется очередь из состояний, которая упорядочивается в соответствии со значениями приращения (разности) ценности состояний.

Ниже приведен пример кода, реализующего п.1 алгоритма, обеспечивающего формирование подмножеств состояний-предшественников для каждого из состояний и сохранение их в словаре **predecessor**:

```
# инициализация словаря с элементами в виде
# состояние-ключ:множество его состояний-предшественников
predecessors = {}

# для каждого состояния state
# формируем множество всех его предшествующих состояний
for state in self.mdp.getStates():
    # если состояние не терминальное
    if not self.mdp.isTerminal(state):
        # для всех действий из состояния state
        for action in self.mdp.getPossibleActions(state):
            # для все следующих состояний nextState после (state,action)
            for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action):
                # если след. состояние уже есть в словаре predecessors
                if nextState in predecessors:
                    # то добавляем в множество с ключем nextState его предшественника state
                    predecessors[nextState].add(state)
                else:
                    # иначе множеству с ключем nextState присваиваем начальное значение {state}
                    predecessors[nextState] = {state}
```

Для реализации п. 2 и п.3 алгоритма и первоначального формирования приоритетной очереди состояний можно использовать код:

```
# создаем приоритетную очередь
pq = util.PriorityQueue()
# для каждого состояния state
for state in self.mdp.getStates():
    # если состояние не терминальное
    if not self.mdp.isTerminal(state):
        values = []
        # для всех действий из состояния state
        for action in self.mdp.getPossibleActions(state):
            # вычисляем q-ценность пары (state, action)
            q_value = self.computeQValueFromValues(state, action)
            # складываем q-ценности в списке values
            values.append(q_value)
```

```

# находим абсолютное значение разности между
#текущим значением ценности state
# в self.values и наивысшим значением Q для всех возможных действий из state
diff = abs(max(values) - self.values[state])
# обновляем приоритет state в очереди
pq.update(state, - diff)

```

Остальную часть задания, связанную с приоритетным обновлением ценности состояний можно легко реализовать, выполнив адаптацию кода, приведенного выше, с учетом алгоритма, изложенного в самом задании. При реализации п. 4.3 и п.4.4.1 необходимо вычислять значения ценностей состояний `pq.pop()` и `p`. Рекомендуется для этого вычислять q-ценности с помощью вызова `q_value = self.computeQValueFromValues(state, action)` аналогично тому, как это реализовано в коде выше.

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

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

При реализации метода `computeValueFromQValues(self, state)` необходимо для всех легальных действий `action` в состоянии `state` вычислить ценности q-состояний с помощью `getQValue(state, action)` и вернуть максимальную ценность.

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

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

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

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

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

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

При написании кода метода `getQValue(self, state, action)`, возвращающего аппроксимированное значение $Q(s,a)$, необходимо получить вектор признаков q-состояний с помощью вызова `self.features = self.feateXtractor.getFeatures(state,`

action). А затем для всех признаков i из **self.features[i]** найти взвешенную сумму признаков в соответствии с (5.25).

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

5.5. Содержание отчета

Цель работы, описание основных понятий MDP и уравнений 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, код реализованных агентов и функций с комментариями в соответствии с заданиями 1-10, результаты игр на разных полях игры, их анализ, результаты автооценивания заданий, выводы по проведенным экспериментам с разными алгоритмами обучения с подкреплением.

5.6. Контрольные вопросы

- 5.6.1 Объясните, что понимают под недетерминированными задачами поиска?
- 5.6.2. Объясните основные понятия Марковского процесса принятия решений.
- 5.6.3. Что понимается под функцией политики?
- 5.6.4. Что понимается под функцией полезности?
- 5.6.5. Какая задача называется эпизодической?
- 5.6.7. Запишите выражение для вычисления функции полезности для продолжающихся задач.
- 5.6.8. Как определяется функция ценности состояния?
- 5.6.9. Как определяется q-функция ценности состояния-действия?
- 5.6.10. Объясните, как строится MDP дерево поиска?
- 5.6.11. Запишите и объясните уравнение Беллмана для функции ценности состояния.
- 5.6.12. Сформулируйте и объясните алгоритм итерации по значениям ценности состояний.
- 5.6.13. Сформулируйте и объясните метод извлечения политики.
- 5.6.14. Сформулируйте и объясните алгоритм итерации по политикам.
- 5.6.15. Объясните основные понятия обучения с подкреплением.
- 5.6.16. Что понимается под исследованиями и эксплуатацией при RL обучении?
- 5.6.17. Объясните обучение с подкреплением на основе модели.
- 5.6.18. Как классифицируются алгоритмы обучения с подкреплением без модели?

- 5.6.19. Объясните алгоритм прямого оценивания (обучение без модели).
- 5.6.20. Объясните алгоритм обучения на основе временных различий.
- 5.6.21. Объясните алгоритм q-обучения.
- 5.6.22. Объясните, что понимают под ϵ -жадной стратегией.
- 5.6.23. Объясните алгоритм q-обучения с аппроксимацией.
- 5.6.24. Что понимают под функциями разведки (исследования)?

ЛАБОРАТОРНАЯ РАБОТА № 6 ИССЛЕДОВАНИЕ ДИНАМИЧЕСКИХ СЕТЕЙ БАЙЕСА

6.1. Цель работы

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

6.2. Краткие теоретические сведения

6.2.1. Неопределенность и степени уверенности высказываний

Интеллектуальные агенты почти никогда не имеют доступа ко всем необходимым знаниям 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].

6.2.2. Совместные и условные вероятности

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

$$P(X_1 = x_1, X_2 = x_2, \dots, X_n = x_n) = P(x_1, x_2, \dots, x_n). \quad (6.1)$$

Свойства совместного распределения:

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

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

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

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

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_2 .

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

$$P(x|y) = \frac{P(x, y)}{P(y)}. \quad (6.2)$$

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

$$P(y)P(x|y) = P(x, y). \quad (6.3)$$

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

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

$$P(x_1, x_2, \dots, x_n) = \prod_i P(x_i|x_1 \dots x_{i-1}) \quad (6.4)$$

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**:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \quad (6.5)$$

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

6.2.3. Независимость и условная независимость

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

$$\forall x, y : P(x, y) = P(x)P(y). \quad (6.6)$$

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

Условная независимость – это основная форма представления знаний в условиях неопределенности. 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*:

$$\forall x, y, z : P(x, y|z) = P(x|z)P(y|z), \text{ т.е. } X \perp\!\!\!\perp Y|Z \quad (6.7)$$

или, эквивалентно, если и только если

$$\forall x, y, z : P(x|z, y) = P(x|z). \quad (6.8)$$

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

6.2.4. Байесовские сети

Байесовская сеть представляет совместную вероятность множества n дискретных случайных переменных $X_1, X_2, \dots, X_n$ в форме направленного ациклического графа. Каждая вершина графа представляется случайной переменной, с которой связана таблица условных вероятностей, содержащая вероятность каждого состояния переменной с учетом её родителей в графе. Ребра графа обозначают взаимодействия переменных. Интуитивный смысл стрелки вдоль ребра сети Байеса указывает на то, что вершина X оказывает непосредственное влияние на вершину Y [3, 9].

Произвольная вершина X сети Байеса описывается локальным условным распределением:

$$P(X|a_1 \dots a_n), \quad (6.9)$$

где $a_1 \dots a_n$ – родительские (*parents*) вершины для вершины X .

Сеть Байеса неявно кодирует совместное распределение своих переменных как произведение локальных условных распределений

$$P(x_1, x_2, \dots, x_n) = \prod_{i=1}^n P(x_i | \text{parents}(X_i)) \quad (6.10)$$

Утверждения локальной условной независимости можно проверить (верифицировать) непосредственно с помощью структуры сети, используя критерий, называемый **D-разделенностью** [2, 3, 9].

Представление предметной области в виде модели байесовской сети позволяет осуществлять вероятностные логические выводы. **Вероятностный вывод** в сетях Байеса предполагает вычисление вероятностей переменных запроса через вероятности других известных переменных. Известны алгоритмы **точного вероятностного вывода** (например, вывод с использованием метода прямого (полного) перебора или метода исключения переменных), которые характеризуются большой вычислительной сложностью. Точный вероятностный вывод в больших многосвязных сетях, часто является неосуществимым. Поэтому были разработаны **методы приближенного вероятностного вывода**, основанные на формировании случайных выборок из распределений. Вероятностный вывод на основе выборок происходит быстрее, чем вычисление ответа на запрос, например, методом исключения переменных. Точность вывода зависит от количества формируемых выборок.

6.2.5. Формирование выборки из заданного распределения

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 задаваемому распределением вероятности.

Пример. Пусть задано распределение цветов (таблица 6.1), которое в памяти можно хранить в виде словаря с набором пар $\{C: P(C)\}$.

Таблица 6.1. – 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 $C = \text{blue}$. После формирования большого числа выборок можно получить набор выборочных значений с вероятностями, сходящимися к заданному распределению.

6.2.6. 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 состояние погоды в i -ый день. Модель Маркова для примера погоды изображена на рисунке 6.1.

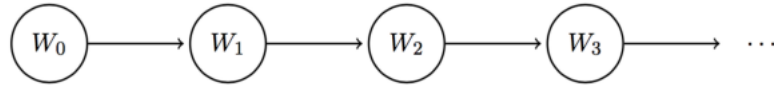


Рисунок 6.1 — Модель Маркова

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** $W_{i+1} \perp\!\!\!\perp \{W_0, \dots, W_{i-1}\} | W_i$. Это позволяет восстановить совместное распределение для первых $n + 1$ переменных с помощью цепочного правила следующим образом:

$$Pr(W_0, W_1, \dots, W_n) = Pr(W_0)Pr(W_1|W_0)Pr(W_2|W_1)\dots Pr(W_n|W_{n-1}). \quad (6.11)$$

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

$$Pr(W_{i+1}) = \sum_{w_i} Pr(w_i, W_{i+1}). \quad (6.12)$$

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

$$Pr(W_{i+1}) = \sum_{w_i} Pr(W_{i+1}|w_i)Pr(w_i). \quad (6.13)$$

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.

6.2.7. 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 6.2 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 состояниями** 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.

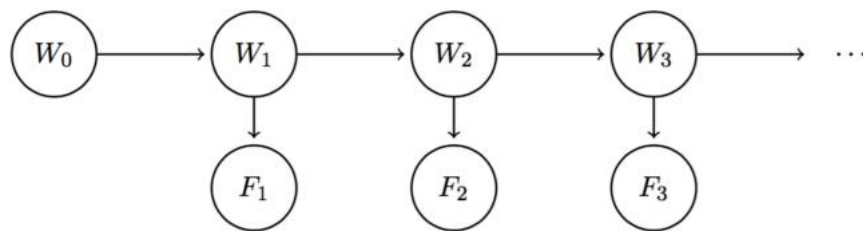


Рисунок 6.2 – Скрытая модель Маркова

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

$$\begin{aligned}
 & F_1 \perp\!\!\!\perp W_0 | W_1, \\
 & \square i = 2, \dots, n \quad W_i \perp\!\!\!\perp \{ W_0, \dots, W_{i-2}; F_1, \dots, F_{i-1} \} | W_{i-1}, \\
 & \square i = 2, \dots, n \quad F_i \perp\!\!\!\perp \{ W_0, \dots, W_{i-1}; F_1, \dots, F_{i-1} \} | W_i.
 \end{aligned} \tag{6.14}$$

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 возмож-

ных значений состояний W_i по всем свидетельствам $f_1, \dots, f_i$, поступившим к моменту времени i как:

$$B(W_i) = Pr(W_i | f_1, \dots, f_i). \quad (6.15)$$

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

$$B'(W_i) = Pr(W_i | f_1, \dots, f_{i-1}) \quad (6.16)$$

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

$$B'(W_{i+1}) = Pr(W_{i+1} | f_1, \dots, f_i) = \sum_{w_i} Pr(W_{i+1}, w_i | f_1, \dots, f_i). \quad (6.17)$$

Перепишем соотношение с использованием цепочного правила (6.4):

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

Так как $Pr(w_i | f_1, \dots, f_i) = B(w_i)$ и $W_{i+1} \perp\!\!\!\perp \{f_1, \dots, f_i\} | W_i$, 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 $B'(W_{i+1})$

$$B'(W_{i+1}) = \sum_{w_i} Pr(W_{i+1} | w_i) B(w_i). \quad (6.18)$$

Найдем связь между $B'(W_{i+1})$ и $B(W_{i+1})$. Из правила Байеса (6.5) следует:

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

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

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

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

$$B(W_{i+1}) \propto Pr(f_{i+1} | W_{i+1}) B'(W_{i+1}). \quad (6.19)$$

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 $B'(W_{i+1})$.

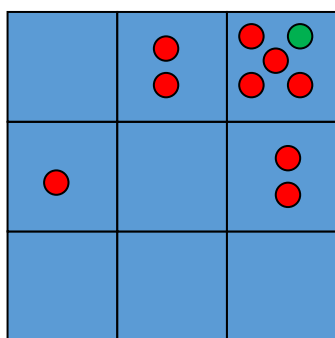
Отметим, что указанный выше трюк с нормализацией может значительно упростить вычисления. Если мы начнем с некоторого начального распределения и будем вычислять распределение степеней уверенности в момент времени t , то можно использовать прямой алгоритм для итеративного вычисления $B(W_1), \dots, B(W_t)$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

6.2.8. 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].

Применение 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 вероятности (доверий) рассматриваемой случайной величины в требуемый момент времени. Частица в этом случае 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 6.3. 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 6.3 – 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 $Pr(T_{i+1} | t_i)$.

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]. 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, следующим образом: для определенного состояния температура может либо оставаться прежней, либо измениться на один градус в диапазоне [10, 20]. Из возможных результирующих состояний 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 этого списка ($T_i = 15$), нам нужна соответствующая модель перехода:

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

Для формирования выборки для частицы в состоянии $T_i = 15$ воспользуемся алгоритмом, описанным в разделе 6.2.5, 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 $0.1 \leq r < 0.9$. Следовательно, в следующий момент времени с учетом таблицы переходных вероятностей частица будет в состоянии $T_{i+1} = 15$.

Допустим 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 $T_{i+1} = 13$.

6.3. Задания для выполнения

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

Задание 0 (0 баллов). Класс `DiscreteDistribution`

Класс `DiscreteDistribution`, определенный в `inference.py`, используется для работы с дискретными распределениями. Этот класс является разновидностью словаря Python, где ключами являются дискретные элементы 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. 6.2.5.

Задание 1 (2 балла). Вероятность наблюдения

В этом задании 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.

Задание 2 (3 балла). 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 (6.19) при получении нового наблюдения **observation**. 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 3 (3 балла). 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. Наблюдения, 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 (6.18). При этом агент имеет доступ к распределению действий призрака через **self.getPositionDistribution**.

Задание 4 (2 балла). Полное тестирование точного вывода

Теперь, когда Пакман знает, как использовать свои априорные знания о поведении призраков и свои наблюдения, он готов эффективно отслеживать 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 5 (2 балла). Инициализация приближенного вывода

В нижеследующих заданиях (5,6 и 7) 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** час-

тицы 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 6 (3 балла). Приближенный вывод: обновление на основе наблюдения

Необходимо реализовать метод **observUpdate** класса **ParticleFilter** в файле **inference.py**. Метод осуществляет обновление на основе наблюдения в соответствии с алгоритмом, описанным в п. 6.2.8. Наблюдение - это зашумленное манхетенское 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 7 (3 балла). Приближенный вывод: обновление во времени

Реализуйте метод **elapsedTime** 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. 6.2.8.

Задание 8 (1 балл). Инициализация при совместной фильтрации частиц

В задании рассматривается случай, когда имеется несколько призраков. Поскольку модели перехода призраков больше не являются независимыми, все призраки должны отслеживаться совместно с использованием динамической сети Байеса (ДСБ), которая является обобщением СММ.

ДСБ с двумя призраками (a и b) изображена на рисунке 6.4. На рисунке скрытые переменные G представляют положения призраков, а переменные свидетельств E представляют собой зашумленные расстояния до каждого из призраков. Представленную структуру ДСБ можно распространить на большее количество призраков.

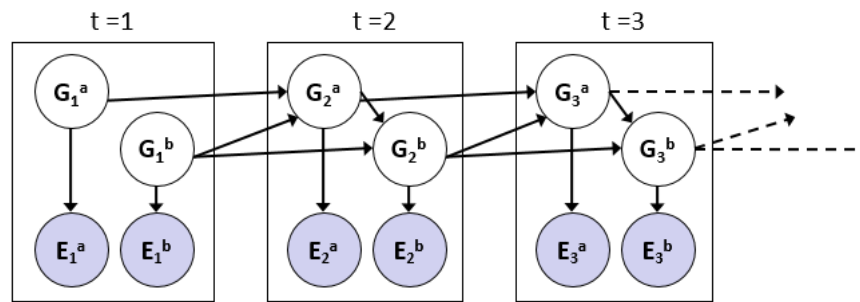


Рисунок 6.4 — Динамическая сеть Байеса с несколькими призраками

В заданиях ниже необходимо реализовать алгоритм вывода, основанный на фильтрации частиц, который одновременно отслеживает несколько призраков. Каждая частица (полная выборка на временном шаге) представляется кортежем позиций призраков, показывающим, где призраки находятся в данный момент. Предоставляемый вам программный код уже подготовлен для извлечения маргинальных распределений по каждому призраку с помощью алгоритма совместного отслеживания призраков, который вы реализуете.

В данном задании завершите определение метода `initializeUniformly` класса `JointParticleFilter` в файле `inference.py`. Метод должен обеспечить начальное равномерное распределение частиц. Как и в задании 5, частицы хранятся в списке частиц `self.particles`.

Задание 9 (3 балла). Обновление на основе наблюдения при совместной фильтрации частиц

В задании необходимо реализовать метод `observeUpdate` класса `JointParticleFilter` файла `inference.py`. Метод должен обеспечивать взвешивание и повторное сэмплирование всех частиц с учетом правдоподобия наблюдаемого расстояния до каждого из призраков. Метод аналогичен одноименному методу класса `ParticleFilter`, но обеспечивает обработку наблюдений для нескольких призраков.

Также реализация метода должна обрабатывать особый случай, когда все частицы получают нулевой вес. В этом случае список частиц `self.particles` следует воссоздать из априорного распределения, вызвав `initializeUniformly`.

Задание 10 (3 балла). Обновление во времени при совместной фильтрации частиц

В задании необходимо завершить определение метода `elapseTime` класса `JointParticleFilter` в файле `inference.py`, чтобы корректно выполнять ресэмплирование частиц в ДСБ совместного отслеживания призраков. В частности, необходимо учитывать, что каждый призрак перемещается в новую позицию, обусловленную позициями всех призраков на предыдущем временном шаге.

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_лаб_6.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.
Файлы, которые необходимо просмотреть	
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.

6.4.3. В рассматриваемом варианте игры цель состоит в том, чтобы выследить невидимых призраков. Пакман оснащен сонаром (слухом), который обеспечивает оценку манхэттенского расстояния до каждого призрака по его шумам. 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, вычисляемых по их шуму. Оценки расстояний, отображаемые в нижней части графического окна, всегда неотрицательны и всегда находятся в пределах 7 единиц от их истинного значения. Вероятность нахождения призрака в соответствующей позиции экспоненциально уменьшается при увеличении отличия от истинного расстояния.

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, СММ 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 динамических байесовских сетей.

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 по одному тесту за раз. 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-ObsProb
```

Как правило, 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**.

6.4.4. В задании 0 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 6.2.5.

Тестов автооценщика 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 док-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 разработанных методов в отчет.

6.4.5. В задании 1 необходимо реализовать метод **getObservationProb(self, noisyDistance, pacmanPosition, ghostPosition, jailPosition)**, который возвращает вероятность наблюдения **noisyDistance** для заданных позиций Пакмана и призрака. Данный метод соответствует 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.

Чтобы протестировать свой код, запустите автооценщик для этого задания:

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

Внесите код и результаты тестирования метода в отчет.

6.4.6. В задании 2 необходимо реализовать метод **observeUpdate** класса **ExactInference**. Метод обеспечивает вычисления в соответствии с правилом обновления (6.19). 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 1), 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 (6.19) при использовании вероятности наблюдения, вычисляемой в ходе вызова:

```
self.getObservationProb(noisyDistance, pacmanPosition,
                        possibleGhostPos, jailPosition)
```

При этом учтите, что значения $B'(W_{i+1})$ из (6.19) 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 автооценщик для этого задания и визуализировать результат используйте команду:

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

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 q2 --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.

6.4.7. В задании 3 необходимо реализовать метод **elapseTime** класса **ExactInference**. Метод выполняет вычисления в соответствии с правилом обновления во времени (6.18), 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 реализации 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 (6.18), 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 необходимо 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 автооценивания этого задания и визуализации результатов используйте команду:

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

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.

6.4.8. В начальной части кода задания 4 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
```

Чтобы запустить автооцениватель для этого задания и визуализировать результат, используйте команду:

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

Если вы хотите запустить этот тест без графики, вы можете добавить параметр **--no-graphics**.

Внесите код метода **ChooseAction** и результаты тестирования в отчет.

6.4.9. При выполнении задания 5 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 инициализируемых частиц задается конструктором класса **ParticleFilter** и по умолчанию равно **self.numParticles=300**. Допустимые позиции частиц **positions** определяются следующим образом: **positions=self.legalPositions**. Прежде чем сформировать список частиц, определите целое число частиц, приходящихся на одну позицию

```
numP=self.numParticles // len(positions)
```

Затем в цикле, пока длина списка **self.particles** не станет равной числу частиц **self.numParticles**, для каждой допустимой позиции частиц **pos** из **positions** необходимо тиражировать позицию **pos**

```
sublistP=[pos]*numP
```

и расширить общий список частиц на **sublistP**, т.е. **self.particles.extend(sublistP)**.

Метод **getBeliefDistribution** получает список частиц и формирует соответствующее распределение. Поэтому в начале создайте переменную-распределение **beliefDist**, которая является экземпляром класса **DiscreteDistribution**. Затем посчитайте число частиц в каждой позиции:

```
for particle in self.particles:
    beliefDist[particle]=beliefDist[particle]+1
```

Нормализуйте полученное распределение **beliefDist** и верните его в качестве результата.

Чтобы протестировать задание выполните команду:

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

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.10. В задании 6 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 соответствии с алгоритмом обновления на основе наблюдения (см. п.6.2.8) 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 q6
```

или без графики

```
python autograder.py -q q6 --no-graphics
```

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.11. В задании 7 необходимо реализовать в виде метода **elapseTime** класса **ParticleFilter** алгоритм обновления во времени, описанный в п. 6.2.8. Так как в итоге метод должен формировать новый список частиц как выборку из распределения, то создайте в начале экземпляра 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 значений { **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=[elapsedDist.sample() for _ in range(int(self.numParticles))]
```

Данный вариант метода **elapsedTime** позволяет отслеживать призраков почти так же эффективно, как и в случае точного вывода.

Обратите внимание, что для этого задания автооценщик тестирует как метод **elapsedTime**, так и полную реализацию фильтра частиц, сочетающую **elapsedTime** и обработку наблюдений.

Чтобы запустить автооценщик для этого задания и визуализировать результаты используйте команду:

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

Для запуска теста без графики добавьте параметр **--no-graphics**. Внесите код разработанных методов и результаты тестирования в отчет.

6.4.12. В начале выполнения задания 8 определите допустимые позиции призраков с помощью **positions=self.legalPositions**. Затем необходимо будет сформировать кортежи из позиций, которые соответствуют одномоментным возможным расположениям разных призраков (в одной позиции может находиться только один призрак). При этом полезны будут возможности модуля Python **itertools**, который обеспечивает реализацию различных итераторов. В частности, используйте функцию **itertools.product**, чтобы сформировать список декартовых произведений (перестановок) из возможных допустимых позиций для **self.numGhosts** призраков:

```
cartProduct=[i for i in itertools.product(positions, repeat=self.numGhosts)]
```

Список **cartProduct** будет состоять из кортежей в виде перестановок позиций призраков. Однако кортежи не будут следовать в случайном порядке. Чтобы обеспечить их случайный выбор, заполняйте список частиц, выбирая элементы из списка **cartProd** с помощью вызова **random.choice()**:

```
self.particles.append(random.choice(cartProduct))
```

Также, как и в задании 5, заполнение списка частиц выполняйте в цикле пока выполняется условие **len(self.particles) <= self.numParticles**.

Чтобы запустить автооценщик для этого задания и визуализировать результаты используйте команду (или с необязательным параметром **--no-graphics**):

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

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.13. При выполнении задания 9 создайте в начале экземпляра распределения **weightsDist** путем вызова конструктора класса **DiscreteDistribution()**.

По-прежнему позицию Пакмана можно определить с помощью метода **gameState.getPacmanPosition()**, но для определения позиции тюрьмы призрака, используйте **self.getJailPosition(i)**, так как теперь есть несколько призраков, у каждого из которых есть своя позиция тюрьмы.

При реализации метода необходимо для каждой частицы (представляемой кортежем позиций) формировать свой список наблюдений **obs**, в котором следует размещать вероятности наблюдений для каждого из призраков. Поэтому необходимо организовать два вложенных цикла:

```
# для каждой из частиц и для каждого из призраков
for p in self.particles:
    obs=[]
    for i in range(self.numGhosts):
        ...
```

Как и в аналогичном методе для класса **ParticleFilter**, здесь необходимо использовать функцию **self.getObservationProb**, чтобы найти вероятность наблюдения с учетом положения Пакмана, потенциального положения призрака и положения тюрьмы:

```
probObs=self.getObservationProb(observation[i],
                                pacmanPosition, p[i], self.getJailPosition(i))
obs.append(probObs)
```

В соответствии с алгоритмом обновления на основе наблюдения (см. п.6.2.8) находим сумму произведений весов совместных наблюдений для каждой комбинации позиций призраков

```
weightsDist[p]+=prod(obs)
```

Нормализуйте полученное распределение **weightsDist** и сформируйте новый список частиц, сделав выборки из **weightsDist** аналогично заданию 6 (см. п.6.4.10).

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 автооценщик для этого задания и визуализировать результаты тестирования, выполните команду (возможно с необязательным параметром **--no-graphics**)

```
python autograder.py -q q9
```

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.14. При выполнении задания 10, как и в задании 9 следует организовать два вложенных цикла по всем частицам и по всем призракам. Вам предлагается следующий шаблон кода, который нужно дополнить:

```

newParticles = []
for oldParticle in self.particles:
    newParticle = list(oldParticle) # Список позиций призраков

    # цикл обновления всех значений в newParticle
    """ ВСТАВЬТЕ ВАШ КОД СЮДА """

    """*** КОНЕЦ ВАШЕГО КОДА ***"""
    newParticles.append(tuple(newParticle))
self.particles = newParticles

```

Вам необходимо в цикле обновить все значения позиций в **newParticle**. Поэтому в цикле

```

for i in range(self.numGhosts):
    ...

```

полагая, что **i** ссылается на индекс призрака, чтобы получить распределение новых позиций этого призрака, учитывая список (**list(oldParticle)**) предыдущих позиций всех призраков, используйте вызов **self.getPositionDistribution(gameState, list(oldParticle), i, self.ghostAgents[i])**. Получив распределение новых позиций призрака, сформируйте выборку из распределения с помощью метода **sample()** и сохраните результат в **newParticle[i]**.

Обратите внимание, что выполнение этого задания включает в себя автооценивание как задания 9, так и задания 10. Поскольку эти задания связаны с вычислением совместного распределения, для их выполнения потребуется больше времени.

При запуске автооценщика учтите, что тесты **q10/1-JointParticlePredict** и **q10/2-JointParticlePredict** проверяют только реализации обновления во времени, а **q10/3-JointParticleFull** выполняет полное тестирование. Обратите внимание на разницу между тестом 1 и тестом 3. В обоих тестах Пакман знает, что призраки будут двигаться по сторонам игрового поля. Чем отличаются тесты и почему?

Чтобы запустить автооценщик для этого задания и визуализировать результаты, используйте команды:

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

или

```
python autograder.py -q q10 --no-graphics
```

Внесите код разработанных методов и результаты тестирования в отчет

6.5. 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 соответствующих алгоритмов вывода, код реализованных функций с комментариями в соответствии с заданиями 1-10, результаты автооценивания заданий, выводы по проведенным экспериментам с разными алгоритмами вероятностных выводов.

6.6. Контрольные вопросы

- 6.6.1 Объясните, что понимают под степенью уверенности высказываний?
- 6.6.2. Что понимают под совместным распределением случайных переменных? Свойства совместного распределения?
- 6.6.3. Что понимают под событием?
- 6.6.4. Что такое маргинальное распределение? Как его получить из совместного распределения случайных переменных?
- 6.6.5. Запишите формулу условной вероятности x при известном y .
- 6.6.6. Запишите правило произведения для 2-х переменных.
- 6.6.7. Запишите цепочное правило.
- 6.6.8. Запишите правило Байеса и объясните его.
- 6.6.9. Определите понятие независимости 2-х случайных переменных.
- 6.6.10. Определите понятие условной независимости 2-х случайных переменных при заданной третьей переменной.
- 6.6.11. Определите понятие сети Байеса.
- 6.6.12. Запишите выражение полного совместного распределения для сети Байеса и объясните его.
- 6.6.13. Сформулируйте критерии D-разделенности для различных триплетов подсетей Байеса.
- 6.6.14. Что понимают под точным и приближенным вероятностным выводом?
- 6.6.15. Сформулируйте алгоритм формирования случайной выборки из заданного распределения.
- 6.6.16. Приведите пример марковской модели. Какие предположения независимости используют в марковской модели?
- 6.6.17. Определите алгоритм прямого распространения для марковской модели.
- 6.6.18. Определите понятие скрытой марковской модели.
- 6.6.19. Сформулируйте задачу фильтрации для СММ.
- 6.6.20. Какие предположения независимости используют в СММ?
- 6.6.21. Сформулируйте правило обновления во времени и правило на основе наблюдений для СММ.
- 6.6.22. Сформулируйте алгоритм прямого распространения для СММ.
- 6.6.23. Что такое фильтрация частиц применительно к СММ?
- 6.6.24. Сформулируйте правило обновления во времени и правило на основе наблюдений, применяемые при фильтрации частиц.

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. — 864 с. — Текст : непосредственный.
3. Рассел, С. Искусственный интеллект: современный подход, 2-е изд. / С. Рассел, П. Норвиг : пер. с англ. — М. : Вильямс, 2006. — 1408 с.
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.

**ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ PYTHON
ДЛЯ СИСТЕМ
ИСКУССТВЕННОГО ИНТЕЛЛЕКТА**

*Методические указания
к лабораторным работам по дисциплине*

*Составитель : **Бондарев** Владимир Николаевич*

В авторской редакции

Изд. № 76/2024. Объем 8,5 п.л. Усл. печ. л. 7,90. Уч.-изд. л. 8,33.
Издательский центр ФГАОУ ВО «Севастопольский государственный университет»