中山大学计算机学院人工智能本科生实验报告

（2023学年春季学期）

课程名称：Artificial Intelligence

教学班级	人工智能	专业（方向）	计算机科学与技术
学号	21307387	**姓名**	**陈雪玮**

一、 实验题目

最短路径搜索、

二、实验内容

1. 算法原理

    1. 读入文本数据，建立两个字典，一个将城市首字母映射为下标，一个将下标映射为城市名；利用下标用邻接矩阵建立一个图，记录图的信息，结点之间的距离；若两个城市之间没有通路，就用无穷大表示。

    2. 根据接收到的要查询起点和终点，对其进行最短路径搜索，这里采用dijsktra算法。基本原理就是从起点出发，广度优先遍历，用贪心的策略，以此访问距离起点最近的结点，并更新起点到其它结点的最短路径（松弛操作），搜索到终点后得到结果。

    3. 根据结果输出最短路径长度和途径结点（即城市，映射回城市名输出即可），并且将结果写入日记文件中。

2. 伪代码

    主函数 lab1_v2_main.py

```python
import Romania_Path

if __name__ == '__main__':
    print('Welcome to inquire the shortest path!')
    while True:
        # 输入查询的两个城市，支持首字母和全称，不分大小写
        city_one = input('Enter the first city:')
        city_two = input('Enter the second city:')
        # 创建Romania对象，使用其中的输入函数format_input输入两个城市，最终会输出结果并且写入日记
        Romania = Romania_Path.Romania()
        Romania.format_input(city_one, city_two)
        # 完成一轮查询后，询问是否要清楚日记和继续
        c = input('Do you want to clear diary? Y/N\n')
        if c == 'Y':
            Romania.clear_diary()
            print('Diary has been cleared!')
        e = input('Do you want to continue inquiry? Y/N\n')
        if e == 'Y':
            continue
        elif e == 'N':
            print('Welcome your next usage!')
            break
        else:
            print('Invalid enter!Continue.')
            continue
```

Romania模块，实现主要功能 Romania.py

```
inf = 最大整形
filename = 要读取的文件名
diary = 写入的文件名
创建类 Romania:
    def 构造函数:（读取文件、生成映射字典、生成图）
        打开filename读取:
            城市数量city_num
            道路数量road_num
            城市之间的距离存到数组city_way

        self.city_graph = 生成city_num个城市的邻接举证
        self.city_title_index = 城市首字母到下标，用于dijsktra算法
        self.city_index_name = 城市下标到城市名，后面用来输出结果
        创建dijsktra算法需要用到的数组
        self.visit = 访问
        self.distance = 距离
        self.path = 前驱

        for way in city_way:  # 生成一个图，这里采用邻接矩阵的方法
            city_one, city_two, city_distance = 读取文件信息，城市一、城市二、两个城
市之间的距离
            city_distance = int(city_distance)
            if 第一个城市首字母 not in self.city_title_index:  # 首字母不在则添加
                city_name = 小写规格化
                city_title = 记录城市的首字母
                self.city_title_index[city_title] = 城市首字母映射为下标
                self.city_index_name[index] = 下标映射为城市名
            第二个城市与第一个城市操作相同
            # 得到两个城市映射的下标，初始化邻接矩阵
            city_one_index = 第一个城市映射后的下标
            city_two_index = 第二个城市映射后的下标
            初始化图
                self.city_graph[city_one_index][city_two_index] = city_distance
                self.city_graph[city_two_index][city_one_index] = city_distance

    def dij(self,start,end)dijskra算法，给出起点到终点，更新访问、距离、前驱数组
        for i in range(self.city_num):
            index = 记录每次访问最近的城市的下标
            self.distance[start] = 0
            寻找距离起点最近的顶点的下标
            for j in range(self.city_num):
                if self.visit[j] == 0 and (index == -1 or self.distance[j] <
self.distance[index]):
                    index = j

            if 找到了终点:
                return
            self.visit[index] = 1
            for j in range(self.city_num):
                if 当前顶点未访问过 and self.distance[index] +
```

```
self.city_graph[index][j] < self.distance[j]:
                    self.distance[j] = self.distance[index] +
self.city_graph[index][j]
                    self.path[j] = index

    def format_output(self, index_one, index_two):格式化输出并且将结果记录到diary中
        对index_one,index_two执行dijsktra算法，得到距离和最短途径城市
        格式化输出并且写入日记

    def format_input(self, city_one, city_two):接受两个城市的输入，并且调用输出数组
给出结果
        识别输入结果，若有效，则映射为城市下标
        self.format_output(index_one, index_two)

    def clear_diary(self):
        清空日记
```

3. 关键代码展示（带注释）

主要是Romania.py

```
import sys

inf = sys.maxsize
filename = 'Romania.txt'
diary = 'Romania_diary.txt'


class Romania:
    def __init__(self):
        with open(filename) as city_data:
            city_num, road_num = city_data.readline().split(' ')
            city_way = city_data.readlines()
            self.city_num = int(city_num)
            self.road_num = int(road_num)
            city_data.close()

        # 对外生成一个城市图 和 一些映射表
        # 因为是已经有的一个图，所以执行dijsktra算法需要的访问数组、距离数组、记录前驱的数组都作为类的成员
        # 这样每次查询的时候就可以利用已经记录的数据，查询速度会快很多
        self.city_graph = list(list(inf for j in range(self.city_num)) for i in range(self.city_num))
        self.city_title_index = dict()  # 城市首字母到下标，用来执行dijsktra算法
        self.city_index_name = dict()  # 城市下标到城市名，后面用来输出结果
        self.visit = list(0 for i in range(self.city_num))  # 访问
        self.distance = list(inf for i in range(self.city_num))  # 距离
        self.path = list(-1 for i in range(self.city_num))  # 前驱

        index = 0  # 作为映射的下标
        for way in city_way:  # 生成一个图，这里采用邻接矩阵的方法
            city_one, city_two, city_distance = way.split()
            city_distance = int(city_distance)
            if city_one.lower()[0] not in self.city_title_index:  # 首字母不再则添加
                city_name = city_one.lower()  # 小写规格化
                city_title = city_name[0]  # 记录城市的首字母
                self.city_title_index[city_title] = index  # 城市首字母映射为下标
                self.city_index_name[index] = city_one  # 下标映射为城市名
                index = index + 1
            if city_two.lower()[0] not in self.city_title_index:
                city_name = city_two.lower()  # 小写规格化
                city_title = city_name[0]  # 记录城市的首字母
                self.city_title_index[city_title] = index  # 城市首字母映射为下标
                self.city_index_name[index] = city_two  # 下标映射为城市名
                index = index + 1
            # 得到两个城市映射的下标，初始化邻接矩阵
            city_one_index = self.city_title_index[city_one.lower()[0]]
            city_two_index = self.city_title_index[city_two.lower()[0]]
            if city_distance < self.city_graph[city_one_index][city_two_index]:  # 初始化图
                self.city_graph[city_one_index][city_two_index] = city_distance
```

```
                self.city_graph[city_two_index][city_one_index] = city_distance

    def dij(self, start, end):  # dijskra算法，给出起点到终点，更新访问、距离、前驱数组
        for i in range(self.city_num):
            index = -1
            self.distance[start] = 0
            for j in range(self.city_num):
                if self.visit[j] == 0 and (index == -1 or self.distance[j] <
self.distance[index]):
                    index = j

            if index == end:  # 如果找到了终点，则退出该函数
                return
            self.visit[index] = 1
            for j in range(self.city_num):
                if self.visit[j] == 0 and self.distance[index] +
self.city_graph[index][j] < self.distance[j]:
                    self.distance[j] = self.distance[index] + self.city_graph[index]
[j]
                    self.path[j] = index

    def format_output(self, index_one, index_two):  # 格式化输出并且将结果记录到diary中
        self.dij(index_one, index_two)
        if self.distance[index_two] == inf:
            print("No way between the two cities!")
            return
        print("The shortest distance is: ", self.distance[index_two])
        print("The shortest path is: ", end='')
        my_path = list()
        index = index_two
        # 将前驱结点依次存入栈，利用栈FILO的性质就可以顺序输出路径了
        while index != -1:
            my_path.append(self.city_index_name[index])
            index = self.path[index]
        with open(diary, 'a') as f:
            f.write("The shortest distance is: " + str(self.distance[index_two]) +
'\n')
            f.write("The shortest path is: ")
            f.close()
        while len(my_path):
            city = my_path.pop()
            print(city, end=' ')
            with open(diary, 'a') as f:
                f.write(city + ' ')
                f.close()
        print('\n')
        with open(diary, 'a') as f:
            f.write('\n')
            f.close()

    def format_input(self, city_one, city_two):  # 用来接受两个城市的输入，并且调用输出数
```

```
组给出结果
        city_one = city_one.lower()[0]
        city_two = city_two.lower()[0]
        # 如果搜索的城市不存在，则返回
        if city_one not in self.city_title_index:
            print("No first city!")
            return
        if city_two not in self.city_title_index:
            print("No second city!")
            return
        index_one = self.city_title_index[city_one]
        index_two = self.city_title_index[city_two]
        self.format_output(index_one, index_two)

    def clear_diary(self):
        with open(diary, 'w') as f:
            f.write('')
```

4. 创新点&优化（如果有）

    1. 增加了清空日记的功能

    ```
    def clear_diary(self):
      with open(diary, 'w') as f:
      f.write('')
    ```

    2. 将dijsktra算法的函数放入类内，将其需要的访问、距离、前驱数组和图都作为类的成员。因为图是不变的，所以在每次计算新的路径时不用重新开始计算更新，可以利用之前查询已有的结果，缩短了函数执行的时间。

    3. dijsktra算法搜索最近顶点可以采用优先队列，可以减少搜索的时间，使搜索时间复杂度由O(n)降为O(lgn)。

三、实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

    Arad → Bucharest(不区分大小写，支持首字母查询)

```
Welcome to inquire the shortest path!
Enter the first city:Arad
Enter the second city:Bucharest
The shortest distance is:  418
The shortest path is: Arad Sibiu RimnicuVilcea Pitesti Bucharest
```

```
Do you want to clear diary? Y/N
N
Do you want to continue inquiry? Y/N
Y
Enter the first city:
```

```
Enter the first city:a
Enter the second city:B
The shortest distance is:  418
The shortest path is: Arad Sibiu RimnicuVilcea Pitesti Bucharest
```

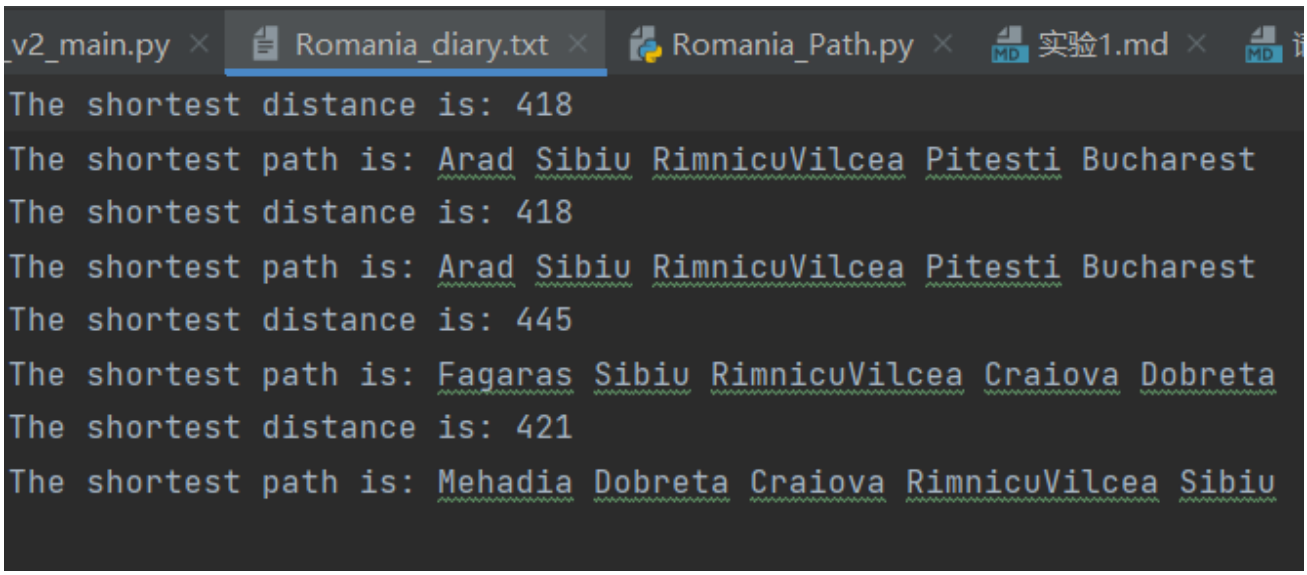
Fagaras → Dobreta

```
Enter the first city:f
Enter the second city:d
The shortest distance is:  445
The shortest path is: Fagaras Sibiu RimnicuVilcea Craiova Dobreta
```

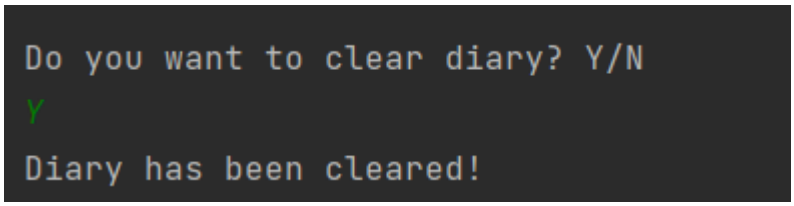
Mehadia → Sibiu

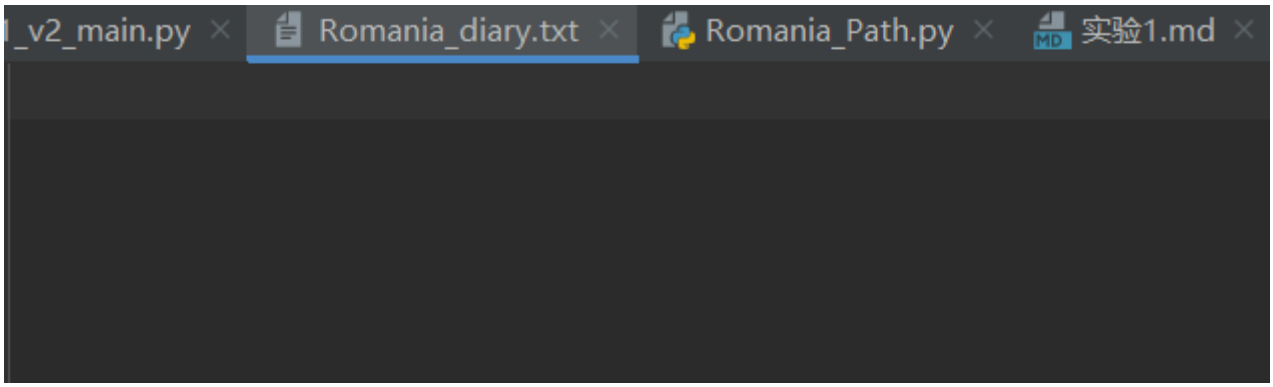
```
Enter the first city:M
Enter the second city:S
The shortest distance is:  421
The shortest path is: Mehadia Dobreta Craiova RimnicuVilcea Sibiu
```

日记记录



清空日记





2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

   运行时间主要是dijsktra算法部分，n个顶点，最多执行n次循环必然访问到终点，每次循环中都需要寻找最近顶点和更新距离，时间复杂度为$O(n^2)$。若采用优先队列搜索最近顶点和存储更新距离，则时间复杂度为O(nlgn)。

|----------**如有优化，请重复1，2，分析优化后的算法结果**----------|

四、思考题

1. 字典的键必须是不可变数据类型。列表是可变数据类型，元组是不可变数据类型，因此用列表作为字典的键编译器会报错，用元组则不会。

2. 不可变数据类型：数字、字符串、元组；

   可变数据类型：列表、字典、集合。

   不可变数据类型

   数字

```
>>> a=1
>>> id(a)
1546211649840
>>> a=2
>>> id(a)
1546211649872
```

   字符串

```
>>> a='ai'
>>> id(a)
1546269051888
>>> a='ex'
>>> id(a)
1546213635312
```

   元组

```
>>> a=(1,2)
>>> id(a)
1546268776448
>>> b=(3,4)
>>> a=a+b
>>> a
(1, 2, 3, 4)
>>>
>>> id(a)
1546269476400
```

   可变数据类型

   列表

```
>>> a=[1,2,3]
>>> id(a)
1546269052608
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> id(a)
1546269052608
```

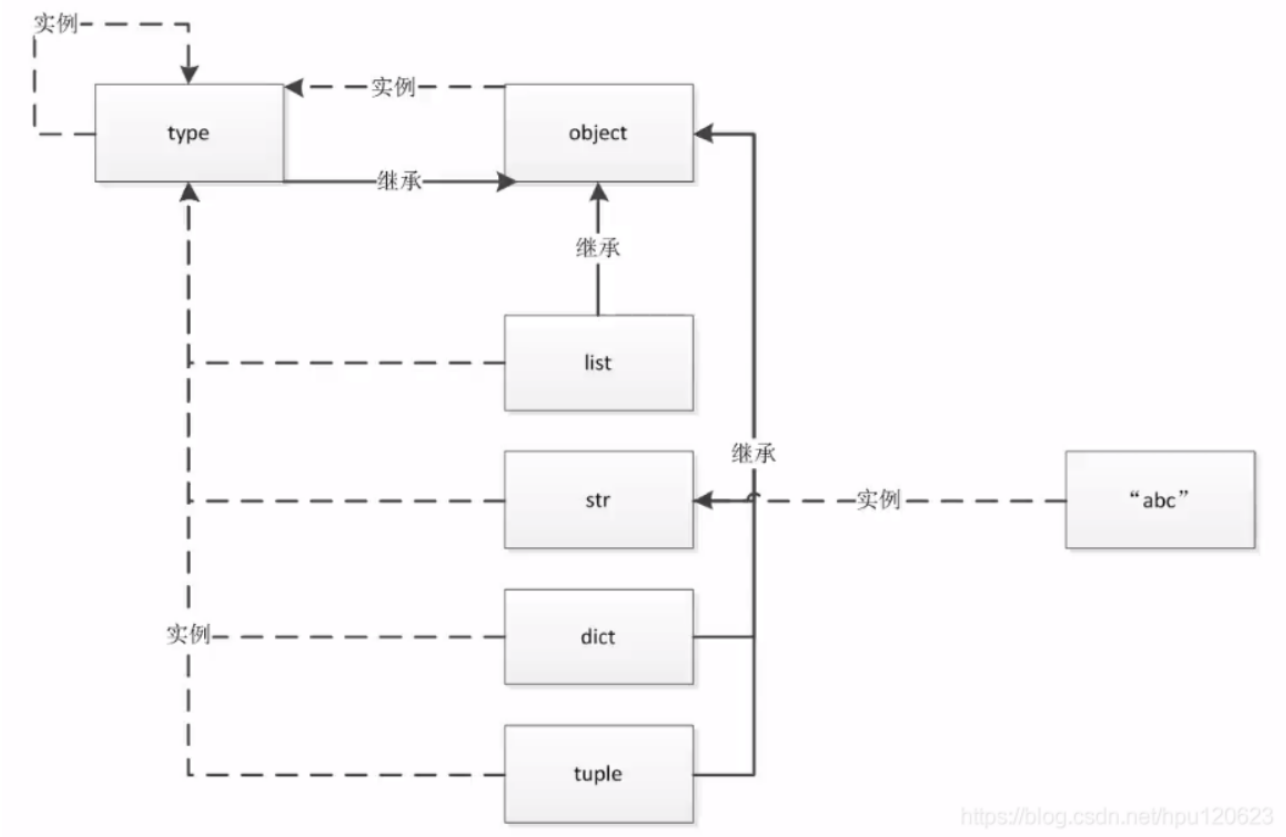
字典

```
>>> a=dict()
>>> a[1]=2
>>> id(a)
1546268419008
>>> a
{1: 2}
>>> a[2]=3
>>> id(a)
1546268419008
>>> a
{1: 2, 2: 3}
```

集合

```
>>> a=set()
>>> a.add('s')
>>> id(a)
1546269270080
>>> a
{'s'}
>>> a.add('uu')
>>> id(a)
1546269270080
>>> a
{'s', 'uu'}
```

3.



1.object是所有数据类型的基类，type、基本数据类型和class类都继承于object。

2.type是object、type、基本数据类型和class类的实例。

3.对类的实例化和基本数据类型的赋值，实际上都是实例化，也就是在创建对象。

综合以上1、2、3点，可以得出，除了object不是对象，它是python中最顶层的基类，它没有父类，其余的所有内容都可以看作是某个类的实例化，也即“一切皆对象”。

五、参考资料

课堂ppt

Python3 教程 | 菜鸟教程 (runoob.com)

彻底搞懂Python一切皆对象！！！_python一切皆对象的理解_火航的博客-CSDN博客