深 圳 大 学 实 验 报 告

课程名称：Python 程序设计

实验项目名称：作业 2

学院：计算机与软件学院

专业：软件工程

指导教师：潘浩源

报告人：郑彦薇 学号：2020151022

实验时间：2022/6/6~2022/6/10

实验报告提交时间：2022/6/10

教务部制

一、实验目的：

利用所学基础知识与数据结构，解决实际问题。

二、实验方法步骤

1、读题，对每个问题提出解决问题的思路

2、对需要编程解决的题目按照得到的思路编写源代码

3、运行代码，调试错误，直到输出正确结果

三、实验过程及内容：

（一）完成课程讲义中状态机的实现

1. 解题思路

➢ **状态机 SM**

1) 父类状态机 SM 的函数编写，主要包括 init 函数和 transduce 函数；而对于函数 transition_fn 以及 output_fn，则在子类中根据相应功能进行具体编写。

2) Init 函数：初始化私有成员 start_state 的值为 None 即可。

3) Transduce 函数：对于每一个子类的不同输入，定义可一致化的函数。访问输入中的每一项，对其进行相应功能操作（即 transition_fn），以输入长度设置循环，不断更新状态值（即 start_state），最后对其进行输出即可。

代码及细节解释如下：

```
class SM:
    def __init__(self):
        self.start_state = None
    def transition_fn(self, s, x):#在子类中实现
        pass
    def output_fn(self, s):#在子类中实现
        pass
    def transduce(self, input_seq):
        for i in range(len(input_seq)):
            self.start_state = self.transition_fn(self, self.start_state, input_seq[i])
        print(self.output_fn(self, self.start_state))
```

根据输入列表长度设置循环，访问列表每一项，进行相应的功能操作，不断更新 state 值

➢ **累加器 Accumulator**

1) Start_state 的初始化：该功能是实现对列表元素的相加，只需记录每次相加的结果，因此在该类中，将 start_state 初始化为 0。

2) Transition_fn 函数：累加器功能是实现对输入列表中每一位元素的相加。根据功能以及父类 transduce 函数方法，只需返回每次执行结果（即列表中第 i 个元素 x 与前面元素求和结果 s 的和）即可。

3) Output_fn 函数：对最终结果进行返回即可。

代码及细节解释如下：

```
class Accumulator(SM):
    start_state = 0
    def transition_fn(self, s, x):
        return s + x
    def output_fn(self, s):
        return s
```

➢ **二进制相加 Binary_Addition**

1) Start_state 的初始化：在进行二进制相加中，除了对当前对应位置值的相加结果进行记录，还需记录进位状态；另外根据例子可以知道，二进制相加结果将每一位作为列表元素以列表形式存储。因此在该类中，将 start_state 初始化为（0，[]），其中 start_state[0]表示进位值，start_state[1]表示当前结果，因为二进制逆序存储，因此直接向结果列表添加元素即可。

2) Transition_fn 函数：

在对当前位置二进制数进行相加时，会产生四种情况：

①当前位置相加结果为 2，且前一个位置相加已发生进位，即进位值与相加结果值的和为 3：当前结果为 1（即结果列表添加元素 1），且仍需向前进一位（即进位值更新为 1）

②当前位置相加结果为 2，且前一个位置相加不发生进位，或当前位置相加结果为 1，且前一个位置相加发生进位，即进位值与相加结果值的和为 2：当前结果为 0（即结果列表添加元素 0），且需要向前进一位（即进位值更新为 1）

③当前位置相加结果为 1，且前一个位置相加不发生进位，或当前位置相加为 0，且前一个位置相加发生进位，即进位值与相加结果值的和为 1：当前结果为 1（即结果列表添加元素 1），且无需向前进位（即进位值为 0）

④当前位置相加结果为 0，且前一个位置相加不发生进位，即进位值与相加结果值的和为 0：当前结果为 0（即结果列表添加元素 0），且无需进位（即进位值为 0）

根据上述四种情况设置四个判断条件，对相应结果状态进行更新即可。

3) Output_fn 函数：首先判断当前结果的进位值是否为 1，为 1 则表示二进制相加结果需要再添加元素 1，否则直接将结果列表进行返回。如 111+111 得到的状态值为（1，110），根据状态值为 1，结果列表添加元素 1，得到最终正确结果：1110。

代码及细节解释如下：

```
class Binary_Addition(SM):
    start_state = (0, [])
    def transition_fn(self, s, x):
        temp = s[1]
        if s[0] + x[0] + x[1] == 3:
            temp.append(1)
            s = (1, temp)
        elif s[0] + x[0] + x[1] == 2:
            temp.append(0)
            s = (1, temp)
        elif s[0] + x[0] + x[1] == 1:
            temp.append(1)
            s = (0, temp)
        else:
            temp.append(0)
            s = (0, temp)
        return s
    def output_fn(self, s):
        if s[0] == 1:
            s[1].append(1)
        return s[1]
```

根据所描述四种情况确定 state 值

最后一步对进位值的判断，得到最终结果

➢ **反向器 Reverser**

1) Start_state 的初始化：根据题目要求，首先需要根据 seq1 的长度输出对应个数 None；其

次需要根据 end 的出现位置停止对 seq1 的读取以及 seq1 长度的更新;接着需要继续读入 seq2，并不断更新最终输出列表长度。因此在该类中，将 start_state 初始化为（0，0，0，[]），其中 start_state[0]的值为 0 或 1，表示 seq1 是否读取完毕；start_state[1]记录 seq1 的长度，作为首先输出多少个 None 的依据；start_state[2]记录需要进行输出的列表总长度，在完成对 seq1 元素的反向输出后，确定仍需输出多少个 None 的依据；start_state[3]记录已输入列表（即根据读取 input_seq 的结果，同步更新该列表）

2) Transition_fn 函数：根据上述所说 state 每一个位置的值所表示的意义，设置判断条件：若 seq1 读取未结束即状态值为 0，则 seq1 长度增加；若当前元素为 end，表示 seq1 读取完毕，修改状态；在整个读取过程中，输入列表的长度不断增加，记录列表也不断更新。

3) Output_fn 函数：根据 transition_fn 函数得到的列表长度，设置循环将对应结果添加到结果列表中：seq1 长度个 None、seq1 元素的反向输出、列表剩余元素（根据要求，均为 None）。

代码及细节解释如下：

```
class Reverser(SM):
    start_state = (0, 0, 0, [])
    def transition_fn(self, s, x):
        temp = s[3]
        temp.append(x)
        if s[0] == 1:
            return 1, s[1], s[2] + 1, temp
        if x == 'end':
            return 1, s[1], s[2] + 1, temp
        else:
            return 0, s[1] + 1, s[2] + 1, temp
    def output_fn(self, s):
        res = []
        for i in range(s[2]):
            if i < s[1]:
                res.append(None)
            elif s[1] <= i <= 2*s[1]-1:
                res.append(s[3][2*s[1]-1-i])
            else:
                res.append(None)
        return res
```

状态值，判断 seq1 是否读取完毕

seq1 列表长度

已输入列表长度

已输入列表，最终与 input_seq 相同

2. 程序运行结果展示

主函数中设定输入值：

```
print("Accumulator:")
A = Accumulator
A.transduce(A, [-1, 2, 3, -2, 5, 6])
print("Binary_Addition:")
B = Binary_Addition
B.transduce(B, [(1, 1), (1, 0), (0, 0)])
print("Reverser:")
R = Reverser
R.transduce(R, list('nehznehS evol I') + ['end'] + list(range(15)))
```

对应输出结果：

```
Accumulator:
13
Binary_Addition:
[0, 0, 1]
Reverser:
[None, None, None, None, None, None, None, None, None, None,
 None, None, None, None, None, 'I', ' ', 'l', 'o', 'v', 'e', ' ',
  'S', 'h', 'e', 'n', 'z', 'h', 'e', 'n', None]
```

（二）纸牌游戏

1. 解题思路

- **Card 类的补充**

根据类中函数的功能，可以知道待补充的三个函数为比较函数，分别进行大于、小于或等于的比较，在类中进行补全即可。

代码及细节解释如下：

```python
    def __gt__(self, other):#比较函数，判断前一张牌是否大于后一张牌
        if not isinstance(other, Card):#非法性判断，判断是否为Card类
            raise TypeError('Illegal Type!')
        return self.rank > other.rank

    def __lt__(self, other):#比较函数，判断前一张牌是否小于后一张牌
        if not isinstance(other, Card):
            raise TypeError('Illegal Type!')
        return self.rank < other.rank

    def __eq__(self, other):#比较函数，判断两张牌大小是否相同
        if not isinstance(other, Card):
            raise TypeError('Illegal Type!')
        return self.rank == other.rank
```

- **Hand 类的补充**

1) receive_cards 函数：玩家收到多张牌时，直接将收到的卡片列表添加到原有的手牌列表中即可。

2) receive_card 函数：玩家收到一张牌时，可以通过列表的 append 功能实现对手牌列表牌的补充。

3) give_card 函数：玩家给出牌前已经通过 shuffle 函数实现“洗牌”，首先判断手牌是否已空，如果已空，则返回 None，否则直接给出第一张牌，同时将该牌从手牌列表中进行删除。

4) give_cards 函数：玩家给出多张牌，只需直接将玩家的手牌进行清空即可。

代码及细节解释如下：

```
    def receive_cards(self, cards):#收到多张卡片
        self.cards = self.cards + cards #直接将收到的卡片补充到当前手牌列表中

    def receive_card(self, card):#收到一张卡片
        self.cards.append(card) #添加单张牌

    def give_card(self):#给出一张牌
        if self.cards:
            temp = self.cards[0]
            del(self.cards[0])
            return temp
        else:
            return None

    def give_cards(self):#给出多张牌
        temp = self.cards.copy()
        self.cards.clear()
        return temp
```

临时变量，记录第一张手牌

将第一张牌从手牌列表中删除

需要使用 copy 复制列表信息，如果只是简单的赋值：temp=self.cards，对 self.cards 操作时，也对 temp 进行了操作，导致错误

- **Deck 类的补充**

进行 52 张牌的分配，只需利用花色和点数设置循环，按照（点数，花色）的格式将牌信息进行添加即可。

代码及细节解释如下：

```
def __init__(self):
    self.cards = []
    for i in ("H", "S", "D", "C"):
        for j in range(2, 15):
            self.cards.append(Card(j, i))
    #self.cards = [Card(2, "H"), Card(2, "S"), Card(3, "D"), Card(3, "S")]  # 仅供测试用，补充上句后可注释
    self.size = self.num_cards
    self.shuffle()
```

- **Player 类的补充**

在该类中只需对玩家及其所持手牌进行初始化即可，注意手牌的初始化为列表的拷贝，需要使用 copy()函数进行深拷贝，否则会出错。（与上述 hand 类中的 give_cards 函数中的 copy 拷贝同理）。

代码及细节解释如下：

```
def __init__(self, name, cards=[]):#对玩家和所持手牌的初始化
    self.name = name
    self.cards = cards.copy()#深拷贝
```

➢ **Game 类的补充**

1) Deal 函数：实现两个玩家的轮流取牌，直到牌堆被取完为止。

2) Turn 函数：该函数的目的是获取每一轮的胜者。首先当两个玩家手中都还有牌时，游戏就仍在继续，因此可以设置循环条件执行每一轮，每一轮的获胜条件是当前玩家出牌大于另一个玩家，那么会有 3 种情况：第一个玩家出牌更大、第二个玩家出牌更大以及两人出牌大小一致；当循环结束时，同样会有 3 种情况：玩家 1 的牌数为 0，玩家 2 获胜；玩家 2 的牌数为 0，玩家 1 获胜；两人牌数都为 0，达成平手。

3) Play 函数：首先进行发牌，接着使用变量 winner 记录每一轮的获胜者并得到最后的获胜者（即最后一轮的获胜者），同时在两人每次出牌后，都会进行一次洗牌，也就是需要使用 shuffle 函数重新进行随机排序。

代码及细节解释如下：

```
def deal(self):
    #补充代码：给两个玩家发牌。每个玩家轮流从牌堆Deck中取出一张牌，每人26张。
    deck = Deck()
    while deck.num_cards != 0:
        self.players[0].receive_card(deck.give_card())
        self.players[1].receive_card(deck.give_card())
```

在牌堆数为 0 之前，给两个玩家轮流发牌

```
def turn(self):
    temp = Hand()
    while self.players[0].num_cards > 0 and self.players[1].num_cards > 0:
        temp.receive_card(self.players[0].give_card())
        temp.receive_card(self.players[1].give_card())
        print(temp, end="")
        #第一个玩家获胜
        if temp.cards[temp.num_cards - 2] > temp.cards[temp.num_cards - 1]:
            self.players[0].receive_cards(temp.give_cards())
            print("=> " + self.players[0].name)
            return self.players[0]
        #第二个玩家获胜
        elif temp.cards[temp.num_cards - 2] < temp.cards[temp.num_cards - 1]:
            self.players[0].receive_cards(temp.give_cards())
            print("=> " + self.players[1].name)
            return self.players[1]
        else:
            print("=> Tie")
            continue
```

借用中间变量 temp 完成每一轮牌的比较

两个玩家手中牌清空前持续执行每一轮

将每一轮的获胜信息返回给 play 函数的 winner

```
if self.players[0].num_cards == 0 and self.players[1].num_cards > 0:
    self.players[1].receive_cards(temp.give_cards())
    print("=> "+self.players[1].name)
    return self.players[1]
elif self.players[1].num_cards == 0 and self.players[0].num_cards > 0:
    self.players[0].receive_cards(temp.give_cards())
    print("=> "+self.players[0].name)
    return self.players[0]
else:
    print("=> Tie")
    return None
```

两个玩家手中牌清空，进行最后赢家的判断

```
def play(self):
    self.deal()  # 发牌
    winner = None
    while self.players[0].num_cards > 0 and self.players[1].num_cards > 0:
        self.players[0].shuffle()
        self.players[1].shuffle()
        winner = self.turn()
    if winner:
        print(winner.name + " wins! ")
    else:
        print("No winner!")
    return winner
```

玩家手中还有牌，洗牌

记录每一轮赢家

2. 程序运行结果展示

- Test_card():

```
test_Card():
cards: [Card(3,'S'), Card(14,'D'), Card(10,'D'), Card(14,'H')]
max: A♢
min: 3♠
position of max card: 1
regular sorted: ['3♠', '10♢', 'A♢', 'A♡']
reverse sorted: ['A♢', 'A♡', '10♢', '3♠']
```

- Test_hand():

```
test_Hand():
hand: ['2♠', 'K♡', '7♣']
num_cards: 3
give card: 2♠
rest of cards: ['K♡', '7♣']
give card: None
```

- Test_deck():

```
test_Deck():
['5♡', '10♢', '8♢', '5♠', '7♣', '10♠', '9♡', '4♡', '3♡', '2♠', '9♠', '8♠', 'A♠', 'K♠', '2♣', '7♠', '2♡', '10♡', '2♢', '3♣', '10♠', '5♣', '9♢',
 '7♡', '8♠', 'J♡', '6♢', '3♠', 'J♠', 'Q♣', '4♣', 'A♡', '8♡', '6♠', 'K♢', 'A♠', '9♣', 'A♢', 'K♡', 'J♣', '7♢', '4♢', '3♢', '4♠', 'Q♢', 'J♢',
 '6♣', '6♡', 'K♣', 'Q♠', '5♢', 'Q♡']
size of deck: 52
give card: 5♡
rest of cards: ['10♢', '8♢', '5♠', '7♣', '10♠', '9♡', '4♡', '3♡', '2♠', '9♠', '8♠', 'A♠', 'K♠', '2♣', '7♠', '2♡', '10♡', '2♢', '3♣', '10♠',
 '5♣', '9♢', '7♡', '8♠', 'J♡', '6♢', '3♠', 'J♠', 'Q♣', '4♣', 'A♡', '8♡', '6♠', 'K♢', 'A♠', '9♣', 'A♢', 'K♡', 'J♣', '7♢', '4♢', '3♢', '4♠',
 'Q♢', 'J♢', '6♣', '6♡', 'K♣', 'Q♠', '5♢', 'Q♡']
```

- Test_player():

```
test_Player():
<'Pam' has ['10♡']>
```

- Test_game():

```
test_Game():
['Q♡', '9♣']=> A
['Q♡', '5♢']=> A
['7♠', '4♡']=> A
['5♢', '4♠']=> A
['Q♢', 'A♡']=> B
['4♡', '10♢']=> B
['J♢', '8♠']=> A
['A♡', '9♡']=> A
['4♡', 'J♡']=> B
['9♢', '3♠']=> A
['3♡', '2♢']=> A
['Q♢', '6♣']=> A
['J♡', '10♠']=> A
['7♠', '10♣']=> B
['J♢', '10♡']=> A
['2♠', '6♢']=> B
['6♢', '8♡']=> B
['6♢', 'Q♣']=> B
['K♢', 'K♣']=> Tie
['K♢', 'K♣', '7♡', '8♠']=> B
['Q♣', 'K♡']=> B
['Q♣', '2♡']=> A
['A♣', '8♢']=> A
['2♠', 'J♣']=> B
['4♢', '6♡']=> B
['2♡', 'A♢']=> B
B wins!
```

（三）SimpleGraph

1. 解题思路

➢ Vertex()类的补充

该类为顶点类，作为图中的顶点数据类型以及边中的起点和终点数据类型，将变量进行初始化即可。

代码及细节解释如下：

```
class Vertex():
    def __init__(self, name, edges=[]):
        self.name = name
        self.edges = edges
```

➢ Edge()类的补充

该类为边类，边中的起点和终点数据类型均为 Vertex 类，因此需要继承 Vertex 类，然后同样将变量进行初始化即可。

代码及细节解释如下：

```
class Edge(Vertex):
    def __init__(self, start, end, cost=1.0):
        self.start = start
        self.end = end
        self.cost = cost
```

➢ SimpleGraph()类的补充

1) Init 函数：使用列表存储顶点信息和边信息，列表元素分别为 Vertex 类和 Edge 类，由于题中需要使用 dfs 求解路径，因此设置访问数组 visit 并初始化为 0.

2) Add_vertex 函数：该函数是往图中添加顶点信息，只需将顶点以 Vertex 类的形式传入函

数，将其添加到列表中即可。

3) Add_edge 函数：该函数需要往图中添加边，首先根据输入初始化边信息，然后将边添加到列表中，再进行邻接表的创建；遍历顶点列表，确定当前边起点，将边终点添加到起点存放邻居的列表 edges 中。

4) Contains_vertex 函数：遍历顶点列表，判断列表中是否存在点的名称与传入函数的字符串相同，若相同，则点存在，返回 True，否则返回 False。

5) Contains_edge 函数：遍历边列表，判断列表中是否存在边起点名称和边终点名称与传入参数相同，若存在相同，则边存在，返回 True，否则返回 False。

6) Get_neighbors 函数：遍历边列表，找到边起点名称与传入参数相同的边，设置临时变量 temp，用于存放这些边的终点名称，返回该列表即可得到顶点 v 的邻居顶点信息。

7) Is_empty 函数：判断图是否为空，只需判断顶点列表和边列表是否为空。

8) Remove_vertex 函数：删除图中的顶点 v，首先需要将 v 从顶点列表中删除，遍历顶点列表，找到名称与传入参数相同的点信息，使用 remove 函数将其中顶点列表中进行删除既可；其次需要将与 v 有关的边删除，遍历边列表，找到边的起点或终点名称与传入参数相同的边，对该边进行删除即可。

9) Remove_edge 函数：删除图中的边，首先需要将边从边列表中删除，只需对应找到边列表中，起点和终点名称与传入参数相同的边，然后进行删除；其次需要对邻接表中的信息进行删除，在顶点列表中找到边的起点，遍历起点的邻居列表，找到边的终点，对起点邻居列表中对应信息进行删除。

10) Is_neighbor 函数：判断两条边是否相邻，只需遍历边列表判断是否有某边的起点和终点都对应传入参数。

11) Is_reachable 函数：该函数目的是查找是否有从 v1 到达 v2 的路径。补充 index 函数用于获得顶点的下标；补充 dfs 函数对图的点进行深度优先遍历，连通的点更新访问值，访问值相同则说明点连通；补充 visitinit 函数对 visit 数组进行初始化。在 is_reachable 函数中，判断是否存在 v1 到 v2 的路径，首先将 v1 的访问值设置为 1，对 v1 进行深度有点遍历，然后判断 v2 的访问值是否发生改变，若发生改变，则说明 v2 与 v1 连通，即存在 v1 到 v2 的路径。

代码及细节解释如下：

```
def __init__(self, verts=[], edges=[]):
    self.verts = verts
    self.edges = edges
    self.visit = [0] * len(verts)#访问数组，长度与顶点列表长度相同，在dfs中使用

def add_vertex(self, v):
    self.verts.append(v) #列表直接添加顶点元素

def add_edge(self, v1, v2):
    e = Edge(v1, v2, 1.0)
    self.edges.append(e)
    for i in self.verts:#创建邻接表
        if e.start.name == i.name:
            i.edges.append(e.end)
```

创建邻接表，由于是有向图，只需“单向”补充

```
def contains_vertex(self, v):
    for i in self.verts: # 遍历顶点列表，查看是否存在顶点v
        if i.name == v:
            return True
    return False

def contains_edge(self, v1, v2):
    for e in self.edges: # 遍历边列表
        if e.start.name == v1 and e.end.name == v2:
            return True
    return False

def get_neighbors(self, v):
    temp = [] # 记录v的邻居
    for i in self.edges:
        if i.start.name == v:
            temp.append(i.end.name)
    return temp

def is_empty(self):
    # 顶点列表、边列表都为空
    if len(self.verts) == 0 and len(self.edges) == 0:
        print("the graph is empty!")
        return
    print("there are still info didn't be clear!")

def remove_vertex(self, v):
    for i in self.edges:
        if i.start.name == v or i.end.name == v:
            self.edges.remove(i)
    for vex in self.verts:
        if vex.name == v:
            self.verts.remove(vex)# 删除顶点v

def remove_edge(self, v1, v2):
    for i in self.verts:
        if i.name == v1:
            for j in i.edges:
                if j.name == v2:
                    i.edges.remove(j)
    for e in self.edges:
        if e.start.name == v1 and e.end.name == v2:
            self.edges.remove(e)

def is_neighbor(self, v1, v2):
    for i in self.edges:
        if i.start.name == v1 and i.end.name == v2:
            return True
    return False
```

临时变量，存放邻居列表

删除顶点 v，与 v 有关的边也要进行删除

找到边(v1, v2)在邻接表中的存储信息，进行删除

```python
def index(self, v):
    for i in range(len(self.verts)):
        if self.verts[i].name == v:
            return i

def is_reachable(self, v1, v2):
    # 判断是否存在路径可以由顶点 v1 到达顶点 v2,使用dfs求解
    self.VisitInit()
    index1 = self.index(v1)
    index2 = self.index(v2)
    self.DFS(index1)
    if self.visit[index2] == 1:
        return True
    return False

def VisitInit(self):
    self.visit = [0] * len(self.verts)

def DFS(self, i):
    if self.visit[i] == 1:
        return
    self.visit[i] = 1
    for j in self.verts[i].edges:
        d = self.index(j.name)
        self.DFS(d)

def clear_all(self):# 清空图，删除所有顶点和所有边
    self.edges.clear()
    self.verts.clear()

def print(self):  图的信息输出函数
    if len(self.verts) == 0 or len(self.edges) == 0:
        print("the graph is empty!")
        return
    temp = []
    for v in self.verts:
        temp.append(v.name)
    print(f'顶点列表：{temp}')
    temp = []
    for v in self.edges:
        temp.append((v.start.name, v.end.name))
    print(f'边列表：{temp}')
```

添加功能，获得顶点 v 的下标

使用 DFS 实现对 visit 的修改，再根据 visit 判断是否存在路径

初始化 visit 数组，在每一次判断中都需进行初始化，避免错误

调用 clear 功能对列表进行清空

2. 程序运行结果展示

设置 test()函数，使用 add_vertex 函数和 add_edge 函数添加顶点和边信息，初始化图信息，同时测试这两个函数的功能。

```
def test():
    # 创建图信息
    #vex = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
    #edge = [('0', '1'), ('1', '2'), ('1', '3'), ('0', '4'), ('1', '5'), ('2', '6'), ('4', '7'), ('3', '6'), ('5', '9'), ('7', '9')]

    MyGraph = SimpleGraph()
    #MyGraph.print()
    # 测试图的功能函数
    print("测试功能1--添加顶点")
    #往图中添加10个点
    MyGraph.add_vertex(Vertex('0'))
    MyGraph.add_vertex(Vertex('1'))
    MyGraph.add_vertex(Vertex('2'))
    MyGraph.add_vertex(Vertex('3'))
    MyGraph.add_vertex(Vertex('4'))
    MyGraph.add_vertex(Vertex('5'))
    MyGraph.add_vertex(Vertex('6'))
    MyGraph.add_vertex(Vertex('7'))
    MyGraph.add_vertex(Vertex('8'))
    MyGraph.add_vertex(Vertex('9'))
    print("当前图信息：")
    MyGraph.print()

print("测试功能2--添加边")
MyGraph.add_edge(Vertex('0'), Vertex('1'))
MyGraph.add_edge(Vertex('1'), Vertex('2'))
MyGraph.add_edge(Vertex('1'), Vertex('3'))
MyGraph.add_edge(Vertex('0'), Vertex('4'))
MyGraph.add_edge(Vertex('1'), Vertex('5'))
MyGraph.add_edge(Vertex('2'), Vertex('6'))
MyGraph.add_edge(Vertex('4'), Vertex('7'))
MyGraph.add_edge(Vertex('3'), Vertex('6'))
MyGraph.add_edge(Vertex('5'), Vertex('9'))
MyGraph.add_edge(Vertex('7'), Vertex('9'))
print("当前图信息：")
MyGraph.print()
```

得到图信息输出如下：

```
测试功能1--添加顶点
当前图信息：
顶点列表：['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
边列表：[]
测试功能2--添加边
当前图信息：
顶点列表：['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
边列表：[('0', '1'), ('1', '2'), ('1', '3'), ('0', '4'), ('1', '5'), ('2', '6'), ('4', '7'), ('3', '6'), ('5', '9'), ('7', '9')]
```

测试相应的功能：

```
print("现在你拥有一张图，开始测试其他函数功能！")
print("测试功能3--判断图中是否存在顶点v")
if MyGraph.contains_vertex('0'):
    print(f'顶点0在图中')
else:
    print(f'顶点0不在图中')
print("测试功能4--判断图中是否存在边(v1,v2)")
if MyGraph.contains_edge('0', '1'):
    print(f'边(0, 1)在图中')
else:
    print(f'边(0, 1)不在图中')
print("测试功能5--获得顶点v的邻居")
print(f'1的邻居有{MyGraph.get_neighbors("1")}')
print("测试功能6--判断图是否为空")
MyGraph.is_empty()
print("测试功能7--删除顶点v")
MyGraph.remove_vertex('0')
print(f'删除顶点0后的图信息为：')
MyGraph.print()
```

```
print("测试功能8--删除边(v1,v2)")
MyGraph.remove_edge('1', '2')
print(f'删除边(1, 2)后的图信息为：')
MyGraph.print()
print("测试功能9--判断顶点v2是否为v1的邻居")
if MyGraph.is_neighbor('1', '3'):
    print(f'顶点3是1的邻居')
else:
    print(f'顶点3不是1的邻居')
print("测试功能10--判断是否存在路径可以由顶点v1到达顶点v2")
if MyGraph.is_reachable('1', '9'):
    print(f'存在路径由1到达9')
else:
    print(f'不存在路径由1到达9')
print("测试功能11--删除所有顶点和边")
MyGraph.clear_all()
print("删除所有顶点和边后再次测试功能6--判断图是否为空，同时验证clear_all正确性")
MyGraph.is_empty()
```

得到运行结果如下：

```
现在你拥有一张图，开始测试其他函数功能！
测试功能3--判断图中是否存在顶点v
顶点0在图中
测试功能4--判断图中是否存在边(v1,v2)
边(0, 1)在图中
测试功能5--获得顶点v的邻居
1的邻居有['2', '3', '5']
测试功能6--判断图是否为空
there are still info didn't be clear!
测试功能7--删除顶点v
删除顶点0后的图信息为：
顶点列表：['1', '2', '3', '4', '5', '6', '7', '8', '9']
边列表：[('1', '2'), ('1', '3'), ('1', '5'), ('2', '6'), ('4', '7'), ('3', '6'), ('5', '9'), ('7', '9')]
测试功能8--删除边(v1,v2)
删除边(1, 2)后的图信息为：
顶点列表：['1', '2', '3', '4', '5', '6', '7', '8', '9']
边列表：[('1', '3'), ('1', '5'), ('2', '6'), ('4', '7'), ('3', '6'), ('5', '9'), ('7', '9')]
测试功能9--判断顶点v2是否为v1的邻居
顶点3是1的邻居
测试功能10--判断是否存在路径可以由顶点v1到达顶点v2
存在路径由1到达9
测试功能11--删除所有顶点和边
删除所有顶点和边后再次测试功能6--判断图是否为空，同时验证clear_all正确性
the graph is empty!
```

测试完毕。

四、作业完成总结

通过本次作业更加熟悉了类函数的编写格式、实现类功能时对功能函数的编写以及私有成员的正确调用，对 python 类以及类的继承有了更深的认识。

指导教师批阅意见：

成绩评定：

指导教师签字：

年　　月　　日

备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。