1 项目说明

1. 本次实验使用 Python 语言，实现了 Bellman-Ford 与 Dijkstra 两种单源最短路径算法。

2. 程序分为普通版和 UI 版。普通版在控制台输入和输出；UI 版则用一个 UI 界面与用户互动，具体使用方法请参见 Readme.md 文件。两个版本核心算法思想与实现没有本质不同，仅仅是函数返回结果的形式有差异。

3. 本实验报告以普通版代码为例进行设计的介绍。

2 设计思路与实现细节

2.1 点类与边类

1. 一个结点需要，还需要。

2. 一条边需要，还需要。

代码 1: 红黑树结点类

```
class RBnode:
        color = BLACK
        key = None
        left = None
        right = None
        p = None
        part = None
        frequency = None

def __init__(self, newKey=None, newPart=None, newFrequency=None):
        self.key = newKey
        self.part = newPart
        self.frequency = newFrequency
```

代码 2: NIL 结点作为红黑树类的成员变量

```
class RBtree:
    root = RBnode()
```

```
3       __nil = RBnode()
```

2.2 边的松弛

1. 旋转用来调整红黑树的结构，分为左旋与右旋。一般用来帮助其他函数的实现。

代码 3: 左旋函数

```
1       def leftRotate(self, x):
2           y = x.right
3           x. right = y.left
4           if(y.left != self.__nil):
5               y.left.p = x
6           y.p = x.p
7           if(x.p == self.__nil):
8               self.root = y
9           elif(x == x.p.left):
10              x.p.left = y
11          else:
12              x.p.right = y
13          y.left = x
14          x.p = y
```

代码 4: 右旋函数

```
1       def rightRotate(self, y):
2           x = y.left
3           y. left = x.right
4           if(x.right != self.__nil):
5               x.right.p = y
6           x.p = y.p
7           if(y.p == self.__nil):
8               self.root = x
9           elif(y == y.p.left):
10              y.p.left = x
11          else:
12              y.p.right = x
13          x.right = y
14          y.p = x
```

2.3 Bellman-Ford 算法

1. 要插入结点 z，就从根节点开始向下找到它要插入的位置，认适当的结点做父亲；然后设置自己的两个孩子为 NIL，设置自己为红色；最后对自己调用辅助程序 insertFixup 函数重新着色和旋转，使红黑树整体的性质得以保存。

2. 当父亲结点为红色时，这是不行的，就需要继续执行。假定父亲是爷爷的左孩子，insertFixup() 函数分三种情况来修正。

3. 如果叔叔为红色，就让父亲和叔叔变黑、让爷爷变红，并前往爷爷结点处继续递归。

4. 如果叔叔为黑色，且我是右孩子，则前往父亲处左旋。这种情况会转化为下一种情况。

5. 如果叔叔为黑色，且我是左孩子，则使父亲变黑，爷爷变红，并在爷爷处右旋。

6. 如果父亲是爷爷的右孩子，以上左右互换。

7. 最后，将根结点染黑。

代码 5: 插入函数

```
    def insertNode(self, z):
        y = self.__nil
        x = self.root
        while(x != self.__nil):
            y = x
            if (z.key < x.key):
                x = x.left
            elif(z.key > x.key):
                x = x.right
            else:
                print("Key", z.key, "conflict", )
                return
        z.p = y
        if(y == self.__nil):
            self.root = z
        elif(z.key < y.key):
            y.left = z
        else:
            y.right = z
```

```
        z.left = self.__nil
        z.right = self.__nil
        z.color = RED
        self.insertFixup(z)

    def insert(self, k, p, f):
        z = RBnode(k, p, f)
        self.insertNode(z)
```

代码 6: insertFixup 函数

```
    def insertFixup(self, z):
        while(z.p.color == RED):
            y = self.uncle(z)
            if(y.color == RED):
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else:
                if(z.p == z.p.p.left):
                    if(z == z.p.right):
                        z = z.p
                        self.leftRotate(z)
                    z.p.color = BLACK
                    z.p.p.color = RED
                    self.rightRotate(z.p.p)
                else:
                    if(z == z.p.left):
                        z = z.p
                        self.rightRotate(z)
                    z.p.color = BLACK
                    z.p.p.color = RED
                    self.leftRotate(z.p.p)
        self.root.color = BLACK
```

2.4 Dijkstra 算法

1. 要删除结点 z，要用到辅助函数 transplant()，用 v 子树来替换 u 子树。

2. 删除操作类似于普通搜索树的删除，分为三种情况进行。但是删除之后要使用 deleteFixup() 函数，通过改变颜色和执行旋转来恢复红黑性质。

3. deleteFixup() 函数分四种情况来修正。当我是父亲的左子时：

4. 如果兄弟红，就染黑它，染红父亲，并在父亲左旋。去新的兄弟处递归执行。

5. 如果兄弟黑，且其两孩子都黑，则染红兄弟。去父亲递归执行。

6. 如果兄弟 w 黑，且其子左红右黑，则使其与其左子交换颜色，并在 w 处右旋。此时进入第四种情况。

7. 取新的兄弟 w。若兄弟之右子红，则染黑父亲，染黑兄弟右子，在父亲处左旋。

8. 如果我是父亲的右孩子，以上左、右互换。

9. 最后，将自己染黑。

代码 7: 删除函数

```
    def delete(self, i):
        z = self.find(i)
        if(z == self.__nil):
            print('Key', i, 'missing')
            return
        else:
            y = z
            yOriginalColor = y.color
            if(z.left == self.__nil):
                x = z.right
                self.transplant(z, z.right)
            elif(z.right == self.__nil):
                x = z.left
                self.transplant(z, z.left)
            else:
                y = self.minimum(z.right)
                yOriginalColor = y.color
                x = y.right
                if(y.p == z):
                    x.p = y
```

```
                else:
                    self.transplant(y, y.right)
                    y.right = z.right
                    y.right.p = y
                self.transplant(z, y)
                y.left = z.left
                y.left.p = y
                y.color = z.color
        if(yOriginalColor == BLACK):
            self.deleteFixup(x)
```

```
    def deleteFixup(self, x):
        while(x != self.root and x.color == BLACK):
            if(x == x.p.left):
                w = x.p.right
                if(w.color == RED):
                    w.color = BLACK
                    x.p.color = RED
                    self.leftRotate(x.p)
                    w = x.p.right
                if(w.left.color == BLACK and w.right.color == BLACK):
                    w.color = RED
                    x = x.p
                else:
                    if(w.right.color == BLACK):
                        w.left.color = BLACK
                        w.color = RED
                        self.rightRotate(w)
                        w = x.p.right
                    w.color = x.p.color
                    x.p.color = BLACK
                    w.right.color = BLACK
                    self.leftRotate(x.p)
                    x = self.root
            else:
                w = x.p.left
                if(w.color == RED):
                    w.color = BLACK
                    x.p.color = RED
```

```
                    self.rightRotate(x.p)
                    w = x.p.left
                if(w.right.color == BLACK and w.left.color == BLACK):
                    w.color = RED
                    x = x.p
                else:
                    if(w.left.color == BLACK):
                        w.right.color = BLACK
                        w.color = RED
                        self.leftRotate(w)
                        w = x.p.left
                    w.color = x.p.color
                    x.p.color = BLACK
                    w.left.color = BLACK
                    self.rightRotate(x.p)
                    x = self.root
        x.color = BLACK
```

代码 8: transplant 函数

```
    def transplant(self, u, v):
        if(u.p == self.__nil):
            self.root = v
        elif(u == u.p.left):
            u.p.left = v
        else:
            u.p.right = v
        v.p = u.p
```

2.5 打印路线、时间与换乘次数

1. 其他函数的实现大体基于以上核心函数，完成了各种功能。具体请参见源代码。

2. 另外还配套了文件读入等辅助函数，帮助实现本 PJ 要求的功能。

3 用户 UI 的设计与实现

1. 用户 UI 基于 python 的 tkinter 库。

2. 创建了输入输出框并绑定函数事件，让 UI 类拥有一棵树的对象，代用户操作这棵树。

3. UI 实现请参见源代码，不加赘述。