

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、边类与图类

1. 一个结点需要保存它自身的信息，包括名字、当前距离源的距离，还需要上一次换乘站、从上一次换乘站到此站点的线路以及换乘次数。另外为了之后算法使用，还需要重载小于号运算符。
2. 一条边需要其起点和终点的引用，还需要知道自身的权重以及自己是几号线。
3. 一个图需要知道自己的点、自己的边，并提供一个边的名字到边在边列表中下标的字典。

代码 1: 结点类构造函数与重载小于号

```
1 class Vertex:
2     def __init__(self, newName):
3         self.name = newName
4         self.d = 10000
5         self.lastChange = None
6         self.lastLine = -1
7         self.changeTime = 0
8
9     def __lt__(self, other):
10        if(self.d != other.d):
11            return self.d < other.d
12        elif(self.changeTime < other.changeTime):
```

```
13         return True
14     else:
15         return False
```

代码 2: 边类构造函数

```
1 class Edge:
2     def __init__(self, newU, newV, newW, newLine):
3         self.u = newU
4         self.v = newV
5         self.w = newW
6         self.line = newLine
```

代码 3: 图类构造函数

```
1 class Graph:
2     def __init__(self):
3         self.vertices = []
4         self.edges = []
5         self.dic = {}
6         self.readIn()
```

2.2 边的松弛

1. 边的松弛是最关键的核心代码。作为边类的成员函数，一条边比较通过本边去终点和当前去终点的情况，从其中选择一种。

代码 4: 边的松弛函数

```
1     def __relaxCore(self):
2         self.v.d = self.u.d + self.w
3         if(self.u.lastLine == self.line):
4             self.v.lastChange = self.u.lastChange
5             self.v.lastLine = self.u.lastLine
6             self.v.changeTime = self.u.changeTime
7         else:
8             self.v.lastChange = self.u
9             self.v.lastLine = self.line
10            self.v.changeTime = self.u.changeTime + 1
```

```
11
12     def relax(self):
13         if((self.u.d + self.w) < self.v.d):
14             self.__relaxCore()
15             return True
16         elif((self.u.d + self.w) == self.v.d):
17             if(self.u.lastLine == self.line):
18                 newChangeTime = self.u.changeTime
19             else:
20                 newChangeTime = self.u.changeTime + 1
21             if(newChangeTime < self.v.changeTime):
22                 self.__relaxCore()
23             return True
24         else:
25             return False
```

2.3 Bellman-Ford 算法

1. 传统的 Bellman-Ford 算法的方法是对所有的边进行 $V-1$ 次松弛。但对于本次 PJ 来说太慢了，于是我们考虑改进。
2. 事实上，在每一轮中，只需要松弛那些以“上一轮有过更新的结点”为起点的边即可。直到没有边可以松弛。
3. 另外，Bellman-Ford 算法函数作为一个图类的函数存在。

代码 5: Bellman-Ford 算法函数

```
1     def bellmanFord(self, name):
2         self.initialize(name)
3         last = []
4         last.append([])
5         last.append([])
6         last[0].append(name)
7         for i in range(0, len(self.dic)):
8             last[1 - (i % 2)] = []
9             for aname in last[i % 2]:
10                 for edge in self.edges[self.dic[aname]]:
```

```
11         if(edge.relax() and edge.v.name not in last[1 - (i % 2)]):  
12             last[1 - (i % 2)].append(edge.v.name)
```

2.4 Dijkstra 算法

1. 传统的 Dijkstra 算法的方法是先将所有点都放入一个优先队列中，然后逐个弹出，处理后放到已处理的列表中。
2. 本程序将其改进为优先队列内起先只有起点，然后在每一轮中，如果松弛过的边的终点不在队列中也不在处理过的列表中，将其加入队列。
3. 另外，Dijkstra 算法函数作为一个图类的函数存在。

代码 6: Dijkstra 算法函数

```
1  def dijkstra(self, name):  
2      self.initialize(name)  
3      a = []  
4      b = []  
5      b.append(self.vertices[self.dic[name]])  
6      while(len(b) > 0):  
7          b.sort()  
8          u = b[0]  
9          b.pop(0)  
10         a.append(u)  
11         for edge in self.edges[self.dic[u.name]]:  
12             if(edge.v not in a):  
13                 edge.relax()  
14                 if(edge.v not in b):  
15                     b.append(edge.v)
```

2.5 打印路线、时间与换乘次数

1. 因为已经在每一站的结点，都保存了上一次换乘站、从上一次换乘站到此站点的线路，所以打印线路是容易的。

2. 另外, 考虑到换乘次数与时间长短不同, 不具有无后效性的动态规划性质, 所以为了正确计算出换乘次数, 在读入线路时将各条地铁线路都补成了完全图。为此只需改变读入图的函数 `readIn()`, 请参见源代码, 不加赘述。

代码 7: 导航, 并打印路线、时间与换乘次数 (以 Bellman-Ford 为例)

```
1  def getRouteContinue(self, endStation):
2      k = self.dic[endStation]
3      now = self.vertices[k]
4      ans = now.name
5      if(now.lastChange is None):
6          ans = "-Line_" + str(self.edges[self.dic[endStation]][0].line) + \
7              "-" + ans
8      while(now.lastChange is not None):
9          if(now.lastChange.lastChange is not None):
10             ans = now.lastChange.name + "-Line_" + \
11                 str(now.lastLine) + "-" + ans
12         else:
13             ans = "-Line_" + \
14                 str(now.lastLine) + "-" + ans
15         now = now.lastChange
16     return ans
17
18 def navigateBellmanFord(self, line):
19     all = line.split()
20     totalChangeTimes = 0
21     totalTime = 0
22     totalRoute = all[0]
23     if(len(all) < 2):
24         print("Please enter at least two stations!")
25         return
26     for i in range(0, len(all)-1):
27         myGraph.bellmanFord(all[i])
28         totalRoute = totalRoute + myGraph.getRouteContinue(all[i+1])
29         totalTime = totalTime + myGraph.getTime(all[i+1])
30         totalChangeTimes = totalChangeTimes + \
31             myGraph.getChangeTimes(all[i+1])
32     print(totalRoute)
33     print("Expected time:", totalTime, "minutes")
34     print("Expected to change:", totalChangeTimes, "times")
```

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 类拥有一个图的对象，代用户操作这幅图。
3. UI 实现代码多但不含算法相关内容，请参见源代码，不加赘述。