旅游规划程序上机报告

陈贝宁 2100011461

张凌睿 2100011455

**一、问题描述与需求分析**

**题目要求**

开发一个旅游管理系统，方便旅客做出假日出行计划。系统功能需求主要包括系统管理员对系统功能的需求以及旅客对系统功能的需求，这两种用户对系统的需求分别包括:

**1. 系统管理员对功能的需求**

1）旅游景点信息管理：对一些重要景点的信息进行维护，包括景点信息的增加、删除和修改。

2）旅游路线信息管理：对旅游的路线信息的维护，包括路线信息的增加、删除和修改。

**2. 旅游客户对功能的需求**

1）提供三种不同方式的景点信息搜索：

a. 按景点名称搜索，用户可以通过输入景点名称包含的关键字进行查找，系统显示查找到的所有景点信息并提供可能的路线以供选择；

b. 按照景点归属地搜索，通过景点归属地，选择想要搜索的地区，并输入景点包含的关键字，显示查找到的所有景点信息（若无关键字输入则给出系统中保存的该地区所有景点信息）；

c. 按照景点类型（如历史文化、现代都市、山区、海景、综合等）搜索，通过选择景点类型，并输入景点包含的关键字，显示查找到的所有景点信息（若无关键字输入则给出系统中该类型所有景点信息）。

2）单一目的地旅游线路查询：根据用户希望前往的旅游目的地以及不同需求（花费最少、最快到达、少换乘等），分别提供从当前位置出发的路线信息。

3) 多目的地旅游线路查询：根据游客希望前往的多个景点目的地以及当前所在位置，为游客规划一条合适的旅游线路，包含其中所有景点。

**题目分析**

题目要求开发一个旅游规划程序，共有两个大要求：1、信息的存储与增删改查；2、线路的查询。

首先，在信息存储方面，我们考虑用一张图来进行存储：用节点存储景点信息，用边存储路线信息。由于同一路线的往返不一定有相同的参数（时间、花费等），这张图应该是一张有向图。信息的增删改查可以直接在图里实现。

其次，线路的查询分为两种：一种是单源单目的地的查询，我们决定用分支界限法进行搜索；另一种是单源多目的地查询，为了简化问题，我们决定遵循临近点原则，利用广度优先不断搜索最近的景点并不断更新目的地序列。

**二、设计**

我们将所有函数进行了封装，并将相似作用的函数放在同一个文件内，以方便之后调用。大致有以下几类函数：

1）**信息管理函数**：用于存储图与节点

2）**搜索函数**：用于信息的检索

3）**路线搜索函数**：用于路线的搜索规划

4）**预处理函数**：用于预先处理缓存数据

5）**主函数**：利用tkinter库实现一个可交互界面

下面将分别介绍这些函数的代码实现。

**三、具体程序实现**

**1）信息管理函数：**

对于景点而言，可以定义一个对应的节点类来存储，**节点类内置名称、归属地、标签等景点信息。**另外，在存储时，每个节点（景点）有一个互不相同的index值，以便之后调用。代码如下：

class View():

def \_\_init\_\_(self,name,belg,index,tag=None):

# 名称，归属地，标签（用列表）

self.name = name

self.belong = belg

self.index = index

if tag is not None:

self.tag = tag

else:

self.tag = []

def add\_tag(self,ad):

# 用列表

self.tag += ad

def \_\_str\_\_(self):

return '景点名称：%-8s归属地：%-8s标签：%s' \

% (self.name, self.belong, self.tag)

其中，\_\_str\_\_函数使得直接调用景点时，能够返回一组给定的文字。

对于一条边，我们采用边结点类来存储，与景点的节点类相似，内置了起点、终点、耗时、花费等。代码如下：

class ENode():

def \_\_init\_\_(self,vi,ui,time,cost,nxt):

# 边：起点，终点，耗时（分钟），花费

self.start = vi

self.end = ui

self.time = time

self.cost = cost

self.next = nxt

我们打算利用一张有向图来整合景点与边的信息。采用临界出边表，表内每个元素都是一个单链表，其表头是一个节点，之后的每个元素都是以它为起点的边。对于景点的增加，直接在出边表后append即可；删除景点时，为了简化问题，只将被删景点所在的元素改为None，并不需要改动出边表的顺序。而对于边的增删改，直接在单链表内进行操作即可。代码如下：（省略了不重要的代码）

class Graph():

def \_\_init\_\_(self):

self.vertex\_num = 0

self.edges = []

self.vertices = []

self.belongs = set()

self.tags = set()

def get\_vertex\_num(self):

return len(self.vertices)

def add\_vertex(self,v):

。。。

def change\_vertex(self,new\_v):

。。。

def get\_vertex\_byid(self,index):

return self.vertices[index][0]

def del\_vertex(self,vi):

self.vertices[vi] = [None, None]

self.vertex\_num -= 1

def add\_edge(self,vi,ui,time,cost):

# 采用邻接出边表

edge = ENode(vi,ui,time,cost,None)

self.edges.append(edge)

p = self.vertices[vi][1]

if p is None:

self.vertices[vi][1] = edge

else:

while p.next is not None:

p = p.next

p.next = edge

def del\_edge(self,edge):

。。。

def get\_out\_edges(self,vi):

# 返回ENode组成的列表

edges = []

p = self.vertices[vi][1]

while p is not None:

edges.append(p)

p = p.next

return edges

**2）搜索函数：**

我们参阅了一些资料，决定采用正则表达式来进行关键词搜索。这里以名字关键字举例。输入关键字，利用re正则库转化为正则表达式，对出边表内所有不为空的节点进行正则匹配。若匹配成功，则加入待选序列中。所有匹配都完成后，对待选序列进行一次相似性排序，即以相似度、匹配起点等排序。之后返回待选序列。代码如下：

def search\_v(name,graph):

if name == '':

return [i for i in range(graph.get\_vertex\_num()) \

if graph.get\_vertex\_byid(i)!=None]

suggestions = []

p = '.\*?'.join(name)

rp = re.compile(p)

for v,\_ in graph.vertices:

if v==None:

continue

match = rp.search(v.name)

if match:

suggestions.append([len(match.group()),match.start(),v.index])

return [x for \_,\_,x in sorted(suggestions)]

返回待选序列后，再在主函数中与所选归属地、标签等进行匹配，就可以得到最终的结果了。

而边的搜索就直接利用单链表的搜索即可。不再赘述。

**3）路线搜索函数：**

这是**本次大作业的核心算法**，没有之一。如前所述，我们采用了分支界限法来实现单源单目标的搜索。**我们采用优先队列PrioQ来存储元素并按特定权值排序。**每次循环从队列顶部取出一个元素（包括权值、边的出发点、边的终点），并把终点的所有出边放入队列中。在这一过程中，我们还要进行**最优性剪枝**，利用mini[i]记录总起点到vi的最小权值，与取出的权值进行比较，如果比较大就放弃这条路。代码如下：

def nearest\_route(graph, start\_vi, dest, key):

# 分支界限法

# 优先队列元素：（权值，出发点，到达点）

viewQ = PrioQ()

viewQ.enqueue(( [0,0,0] , start\_vi, start\_vi))

# route[i] 表示vi的前缀

vnum = graph.get\_vertex\_num()

route = [-1]\*(vnum)

route[start\_vi] = start\_vi

# mini[i] 表示到vi的最小权值

mini = [-1]\*(vnum)

mini[start\_vi] = 0

# 主循环

f = False

while not viewQ.is\_empty():

w, i, j = viewQ.dequeue()

# 最优性剪枝

if mini[j] >= 0 and w[0] > mini[j]:

# 不是最优

continue

route[j] = i

mini[j] = w[0]

# 找到终点

if j in dest:

f = True

end\_vi = j

break

for e in graph.get\_out\_edges(j):

viewQ.enqueue(( [w[i] + weight(e,key)[i] for i in range(3)],\

e.start, e.end))

# 没找到

if not f:

return False

# 返回路径（id）和权值

p = end\_vi

road = [p]

while p != start\_vi:

road.append(route[p])

p = route[p]

road.reverse()

return road, weight\_sort(w, key)

另外，我们采用route[i]记录到达vi的前缀，这样就可以从终点一直往回记录路径了。

为了应对关于不同的key值（时间最少、花费最省、换乘最少）的优先搜索，我们定义了一个**权值变更函数**，将所需要的权值（时间、花费、换乘数三者之一）移到最前面，以方便优先队列的应用。而为了之后的交互页面展示，还需定义一个权值重整函数，根据key值进行权值顺序的重整。

对于多目的地搜索，我们查阅了相关文献[[1]](#footnote-1)，发现**在可接受范围内，我们可以采用临近点原则，利用单目的地的算法找到最近的目的地**（即最先找到的那个），然后再以这个目的地为新的起点进一步搜索，知道所有目的地都被搜索完成，就从最后一个目的地往最开始的起点搜索。而多目的地没有限制key值，我们依照日常经验，采取换乘最少作为key。代码如下：

def multi\_route(graph, start\_vi, dest):

roads = [] ; wt = (0,0,0)

dest\_t = dest.copy()

while dest\_t:

p = nearest\_route(graph, start\_vi, dest\_t, key='CHANGE')

if not p:

return False

road, w = p

roads += road

wt = [wt[i] + w[i] for i in range(3)]

dest\_t.remove(road[-1])

return roads, wt

**4）数据预处理函数：**

由于每次都手动输入大量的数据过于繁琐，**我们采用txt文档存储一些已有的数据，并在开始运行程序的时候自动读入。**利用with open函数打开txt文件，分行读入并添加至图中。代码如下：

def read\_in(graph):

with open('views.txt') as f:

for line in f.readlines():

if line == 'None\n':

graph.vertices.append([None,None])

continue

index, name, belg, \*tag = line[3:].split()

v = View(name, belg, int(index), tag)

graph.add\_vertex(v)

with open('routes.txt') as f:

for line in f.readlines():

if line == 'None\n':

continue

vi,\_,ui,time,cost = line.split()

vi = int(vi[3:]) ; ui = int(ui[3:])

time = float(time)

cost = float(cost)

graph.add\_edge(vi, ui, time, cost)

当然，txt文件要有存储格式。景点的存储格式如下：‘NO.X 景点名称 归属地 标签\n’这里的X是景点的id值，与景点的先后插入顺序有关；标签可以是复数个；以回车结尾。

边的存储格式如下：‘NO.X to NO.Y T C’其中X、Y为起点、终点的id值，T、C为边的耗时、花费，以2位浮点数存储，单位分别是：分钟、元。

以下两张图是存储范例：

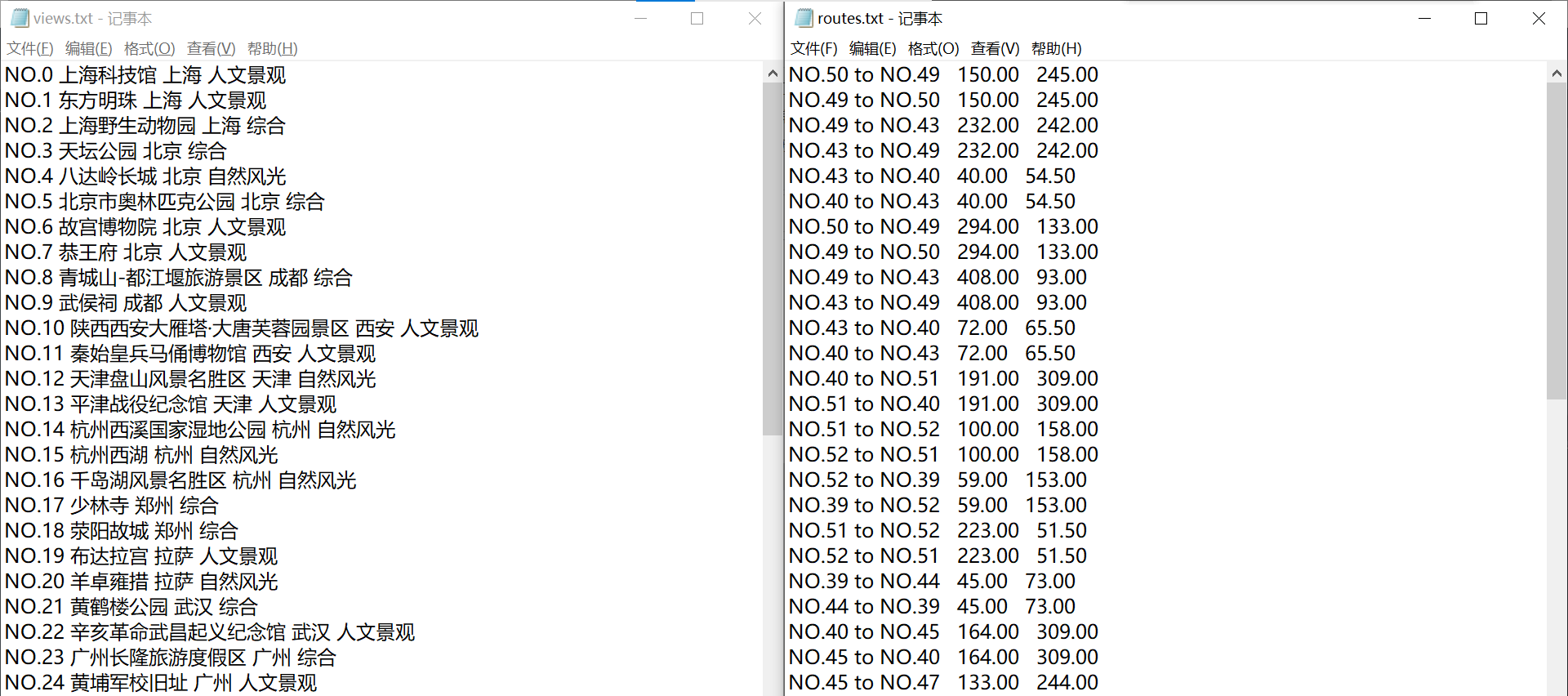


图1. 存储范例

**5）主函数：交互界面**

**我们采用了Tkinter库来开发交互界面。**运行主函数，可以看到如下界面：



图2. 主界面

**1）景点的增加**

对于景点的增删改，我们点击左上角的“景点管理”，会见到如下下拉框：

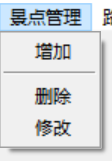


图3. 景点管理下拉框

点击增加按钮，弹出增加景点的弹窗：

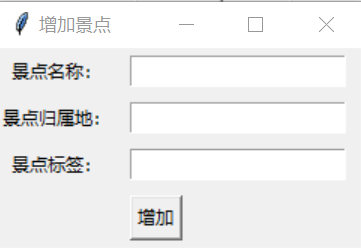
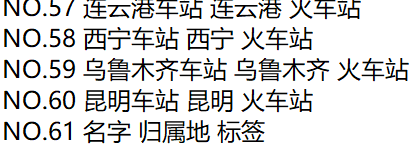
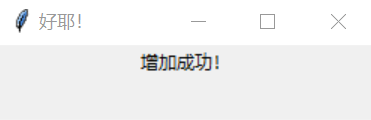
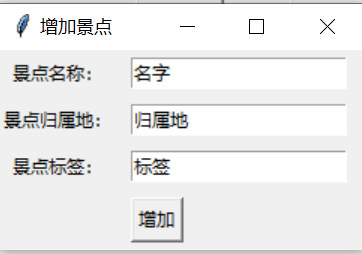


图4. 增加景点弹窗

输入数据即可增加一个新的景点，并在views.txt中增加相应的词条：



组图5. 增加景点成功

**2）景点的删除**

在图3中点击删除，弹出查询景点弹窗：

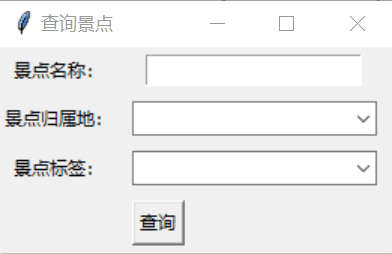
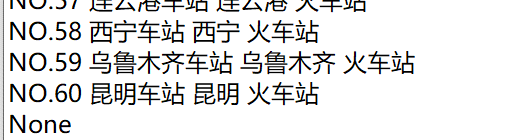
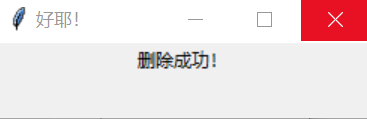
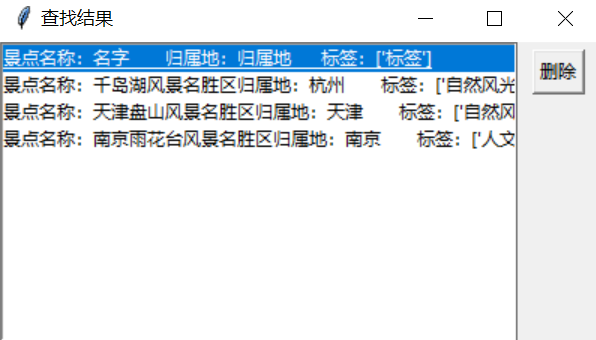
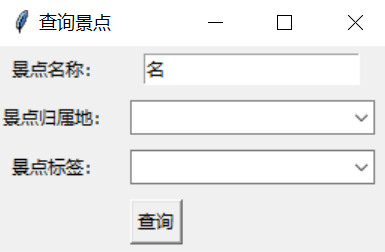


图6. 查询景点弹窗

查询我们想要删除的景点：景点名称支持关键字搜索，归属地与标签支持下拉框选择。填完后点击查询，弹出删除弹窗：

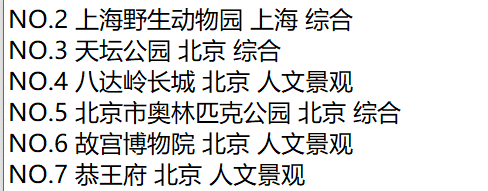
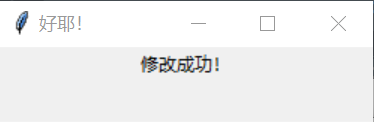
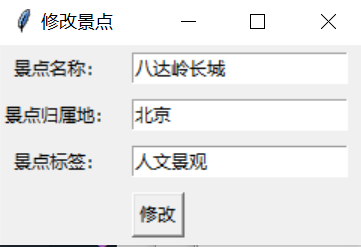
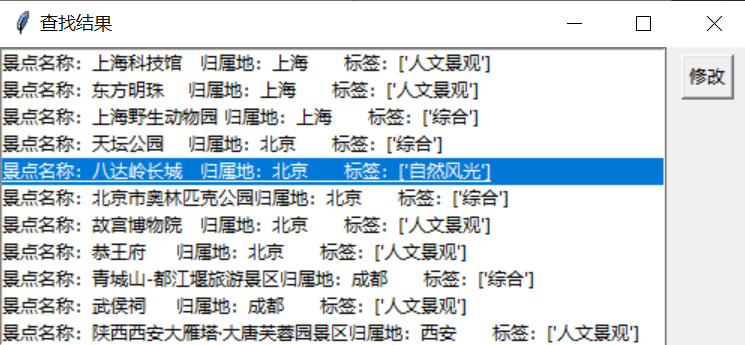


组图7. 删除弹窗

弹出弹窗后可以选择想要删除的景点（只支持删除一个）并点击删除按钮，该景点即可被删除，且在views.txt文档中的相应位置置换为一行None。之后如果再添加一个新景点，id值不会覆盖之前的None（即使是重新添加）。

**3）景点的修改**

在图3中点击修改，弹出图6的查询弹窗。查询后就可以进行修改了，具体操作见图：



组图8. 修改弹窗

如组图8右下图，景点的修改并不会对views.txt中的顺序与id值产生影响，仅会替换景点的信息。

**4）路线的增加**

在主界面（图2）中点击左上角的“路线管理”，可以见到如下下拉框：

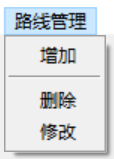


图9. 路线管理下拉框

点击增加按钮，弹出增加路线的弹窗：

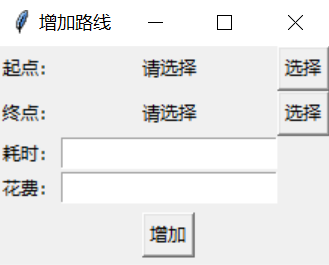
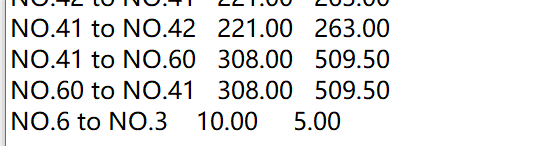
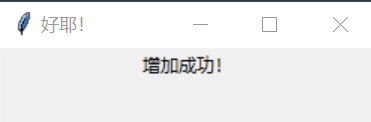
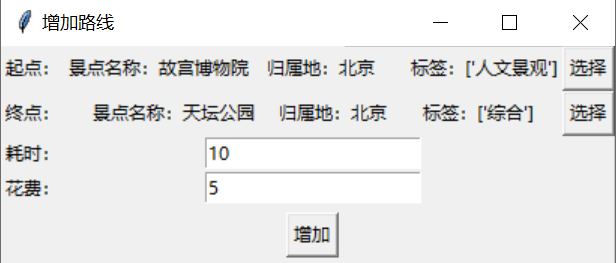


图10. 增加路线弹窗

选择起点与终点，选择过程与景点的查询类似。再输入耗时与花费，即可增加一条新的路线：



组图11. 增加路线过程

如组图11右下图，可以发现routes.txt增加了一条新的路线，起点终点都用景点的id值存储，耗时、花费用2位浮点数存储。

**5）路线的修改**

在图9中点击修改，弹出查找路线的弹窗：

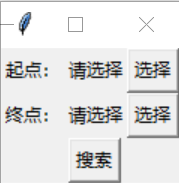


图12. 查找路线弹窗

选择起点终点，并点击搜索，即可返回该路线信息：



图13. 路线信息-修改

特殊地，如果没有这条路线，或者起点终点没有填写完全，会返回报错信息：

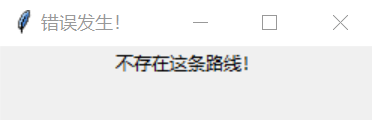
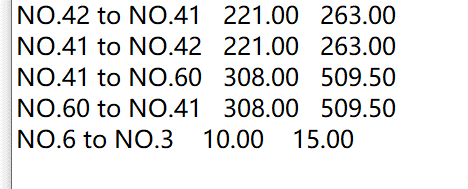
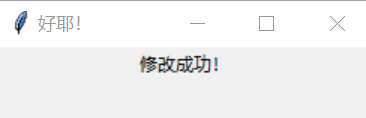
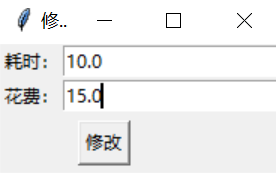
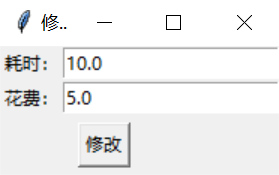


图14. 找不到这条路线

在图13中点击修改，会弹出修改弹窗，即可在弹窗内修改该路线的信息：



组图15. 修改路线过程

可以看到修改路线信息只会对该路线所在行进行覆盖，并不会影响其它行。

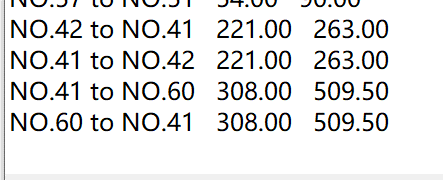
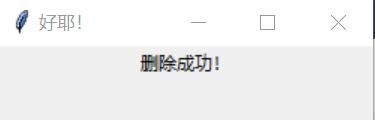
**6）路线的删除**

在图9中点击删除，弹出如图12的查找路线的弹窗。完成起点终点信息的选择后，弹出路线信息的确认弹窗：



图16. 路线信息-删除

点击删除，即可将该路线删除，并在routes.txt中删除该路线所在行，且并不会留下None：



组图17. 删除路线过程

**7）景点与路线的查询**

在主界面（图2）的第一排可以看见两个按钮：“查询景点”、“查询路线”。对于这两个操作我们在之前已经解释过多次，便不再赘述。

**8）完善起点信息**

在刚进入主界面时，会在中央见到如下画面：

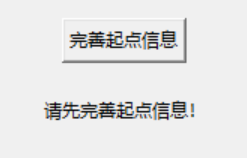


图18. 未完善起点信息

此时如果直接点击第二排的目的地规划，会见到如下报错：

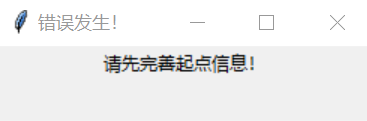


图19. 请先完善起点信息

这时我们需要先完善起点信息。点击主界面中的“完善起点信息”按钮，弹出查询景点弹窗（图6），选择起点（景点中的一个）后，可以见到主界面中已经显示了起点信息：

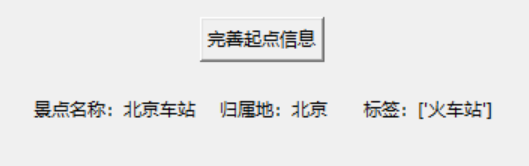


图20. 已完善起点信息

再次点击“完善起点信息”按钮可以更改起点信息。

**9）单目的地规划**

在完善起点信息后，可以进行目的地规划。在主界面（图2）第二排点击“单目的地规划”，弹出以下弹窗：

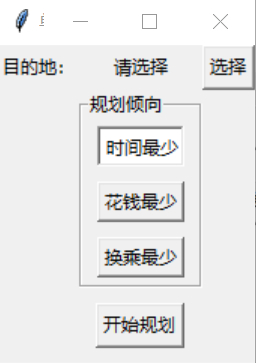
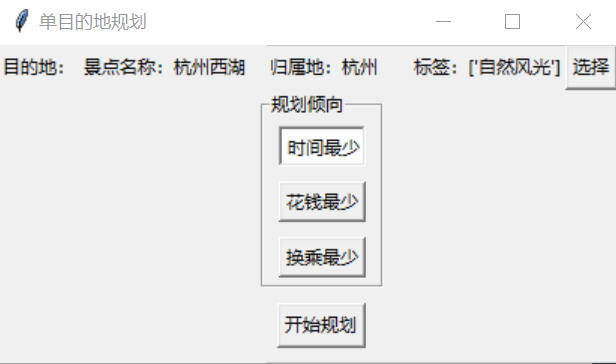
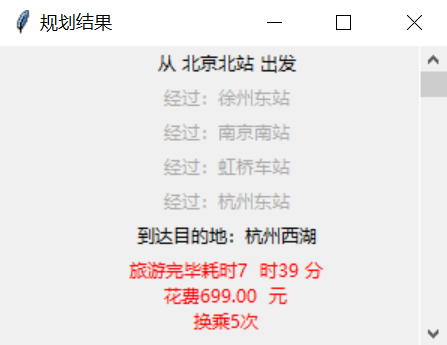
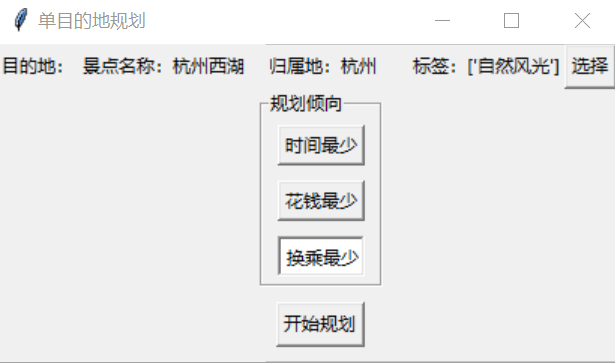


图21. 单目的地规划弹窗

在第一排可以选择目的地，在规划倾向中可以选择一个倾向（时间、花钱、换乘最少）。选择完毕后点击开始规划，即可返回规划结果：





组图22. 不同规划倾向的单目的地规划

这里可以发现时间最少与换乘最少的路线一致，这也在预期当中。另外，路线中经过了许多中间车站，并计算进了换乘次数，这是因为我们精力有限，没有把所有车站两两相连。

**10）多目的地规划**

在主界面（图2）点击第二排的“多目的地规划”，会弹出以下弹窗：

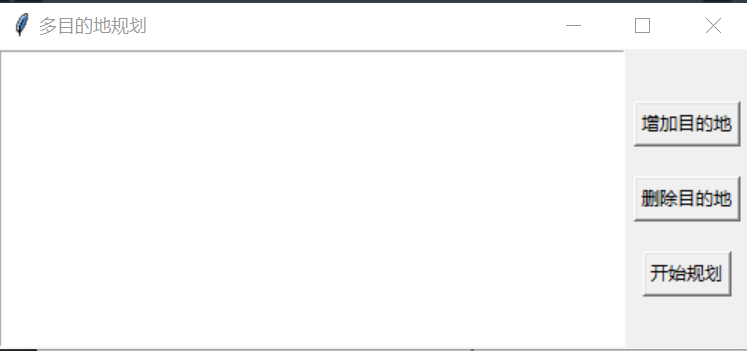


图23. 多目的地规划弹窗

点击增加目的地，可以选择增加一个目的地，总个数不受限制：

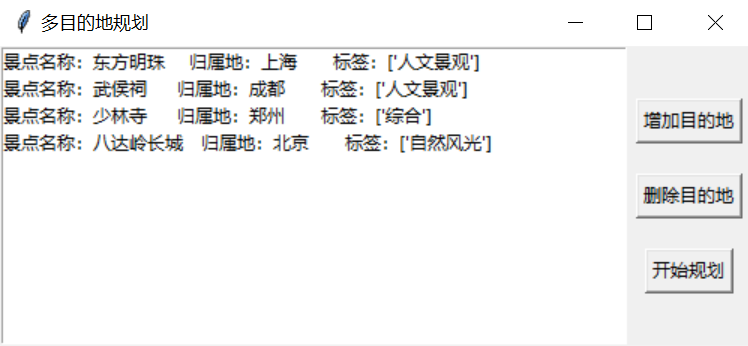


图24. 增加目的地

选择其中一个目的地，点击删除目的地，可以将其从目的地列表中删除：

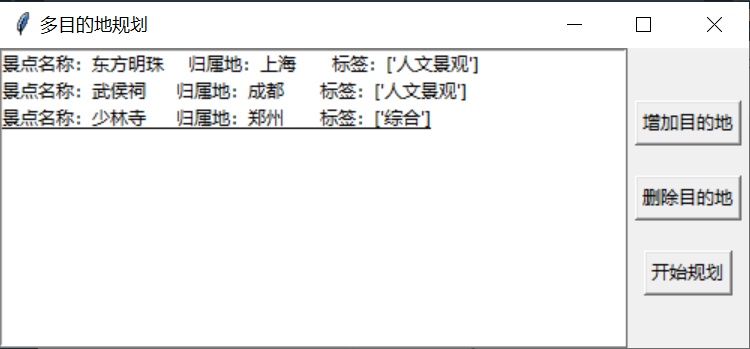
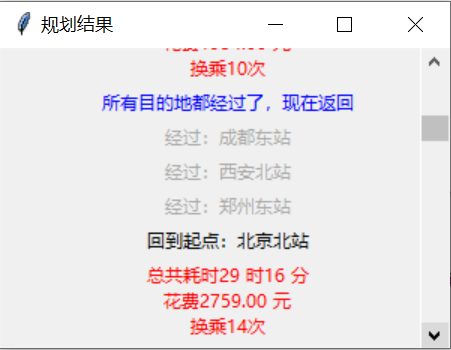


图25. 删除目的地

在选择完所有目的地后，点击开始规划，即可返回规划结果：



组图26. 多目的地规划结果

由于多目的地规划大多时候比较冗长，这里适配了一个滚动条。另外，规划路线也提供了旅游完毕后的返回路径，这一点在之前的单目的地规划也有。

至此，题目的所有要求均已得到实现。

**四、总结**

回顾对课题进行研究探讨的整个过程，我们对本次作业设计的优点与不足总结如下：

**优点：**

1. 利用Tkinter库实现了交互界面，极大地改善了程序使用者的体验；
2. 通过文本管理的方式解决了数据输入的问题，并且可以通过直接编辑文本文档来批量修改数据；
3. 综合运用了若干数据结构预算法。在定义好节点、图的数据类型后，利用分支界限法解决了单目的地的搜索问题，并自行查阅资料解决了多目的地的搜索问题，在实现代码的过程中极大地加深了我们对所学内容的理解。

**不足：**

1. 数据库搭建仍有不足。本次代码配套的数据库中数据组数还是偏少了，导致整个路线图规模不够庞大，无法明显地体现出代码的先进性；为解决这个问题，我们曾试着用网络爬虫获得更多的数据，但却因为格式匹配等问题而未能成功实现。在这一方面，我们还有改进的空间；
2. 交互界面还可进一步改善。在最终版本的程序中，景点还不具备批量添加的功能，也没有直观的单边路线查询的功能。因此，还可以进一步增加交互界面的功能，以提升用户的使用体验。

1. 城市多目的地路径问题的双标号最临近算法, 杨鹤, 《计算机与现代化》, 2008(7), p82-84. [↑](#footnote-ref-1)