项目动机

在使用手机地图软件，查找到达目的地的路线时，软件会精准推荐出发地到目的地的最短路线，并根据实时路况推荐所需事件最短的路线。据此，想要了解使用Python如何能实现这一点。

项目目标

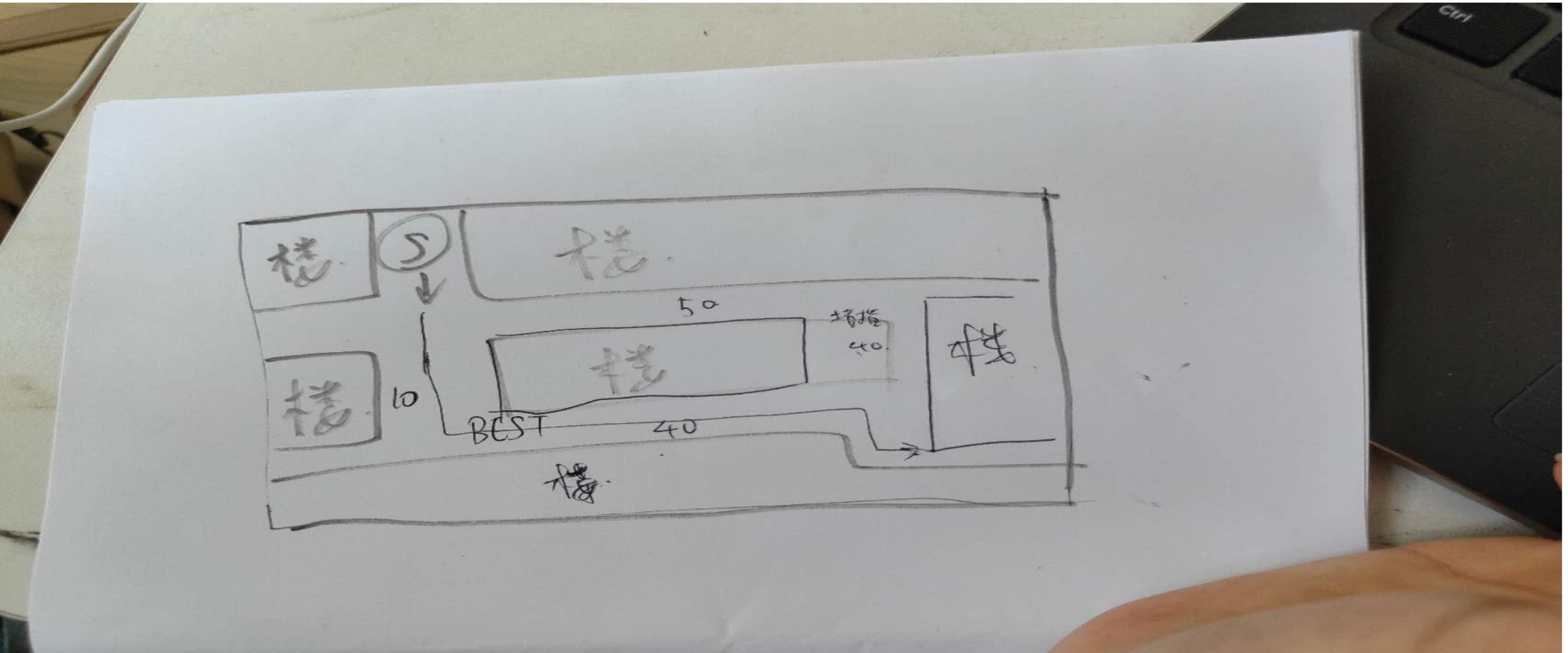
- 目标一：找到最短路线
- 目标二：找到所需时间最短的路线
- 目标三：找到路线最短并所需时间也最短的路线
- 目标一到目标三逐步实现，最终目标是目标三。

项目环境

Windows，Python3，turtle(目标一二中暂时不会用到，在实现目标三时会用)

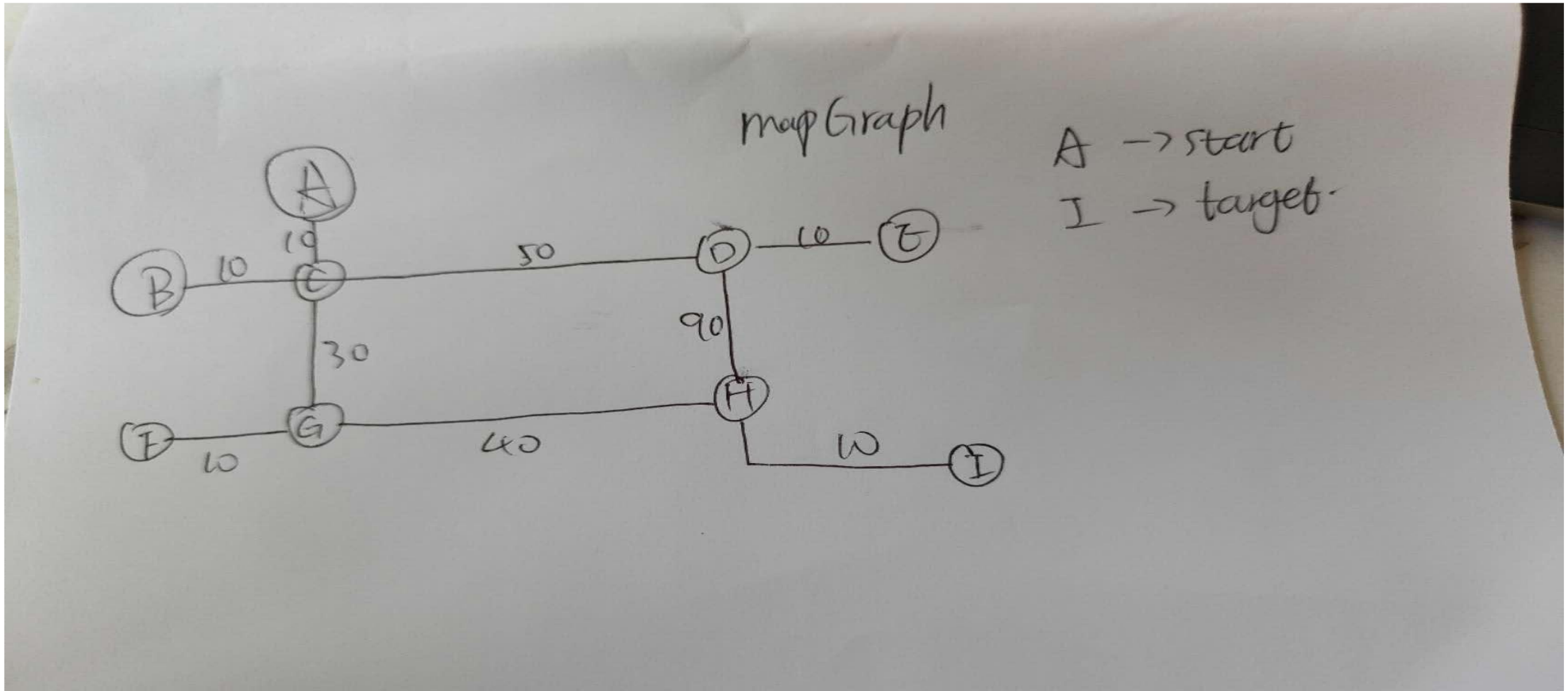
设计目标

- 先实现目标一。如图所示，穿过楼之间的道路，S为起点，道路中间的数字为堵车指数，画出来的路线是最佳路线。



表格理论

- 如图所示，A为起点，I为终点，其他字母为可经过路线中，会经过的节点。每个节点间的数字为堵车指数，堵车指数越低，该段道路越通畅。
- 该图名为mapGraph，顾名思义，后续写代码时，可以用字典类型来表示每个节点之间的关系



已知背景和BFS演示

- 在无方向，有费用的加重值表格当中，从起点出发到终点，找到最便捷的路线，费用越低并路线最短为最佳路线。
- 为方便起见，用字典类型来表示抽象化的表格，也就是将上一页幻灯片的图片代码化。
  - mapGraph={
    ‘A’ :{ ‘C’ :10},
    ‘B’ :{ ‘C’ :10},
    ‘C’ :{ ‘A’ :10, ‘ B’ :10, ‘D’ :50, ‘G’ :30},
    ‘D’ :{ ‘C’ :50, ‘E’ :10, ‘H’ :90},
    ‘E’ :{ ‘D’ :10, ‘H’ :40, ‘I’ :50},
    ‘F’ :{ ‘G’ :10},
    ‘G’ :{ ‘C’ :30, ‘F’ :10, ‘H’ :40},
    ‘H’ :{ ‘D’ :90, ‘E’ :40, ‘G’ :40, ‘I’ :10},
    ‘I’ :{ ‘E’ :50, ‘H’ :10}
  }

目标设计：结果预示和功能介绍

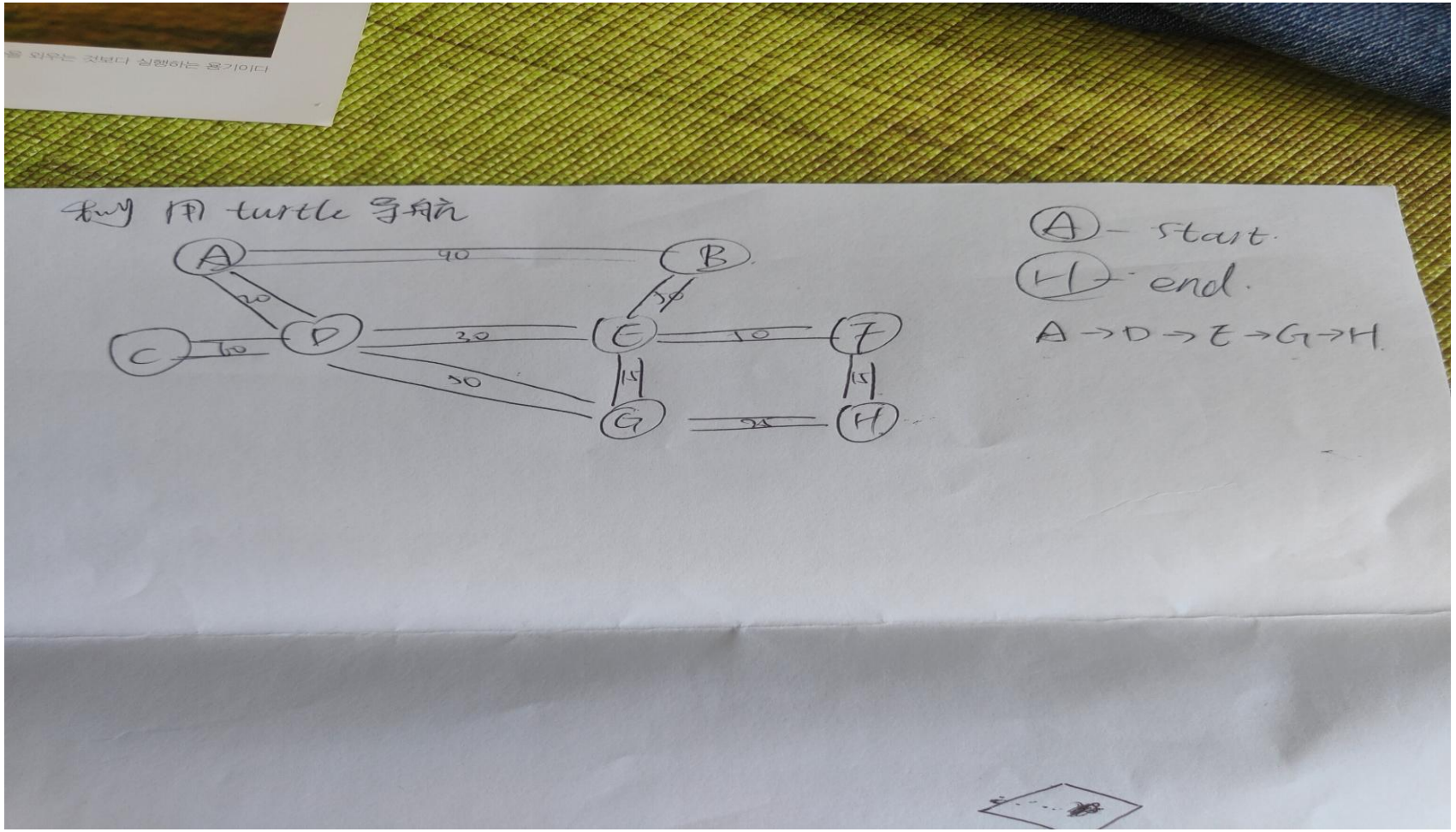
- 结果预示：

```
C:\Users\田园\AppData\Local\Programs\Python\Python313\python.e
path+[node]: ['A', 'C']
path+[node]: ['A', 'C', 'B']
path+[node]: ['A', 'C', 'D']
path+[node]: ['A', 'C', 'G']
path+[node]: ['A', 'C', 'D', 'E']
path+[node]: ['A', 'C', 'D', 'H']
path+[node]: ['A', 'C', 'G', 'F']
path+[node]: ['A', 'C', 'G', 'H']
path+[node]: ['A', 'C', 'D', 'E', 'H']
path+[node]: ['A', 'C', 'D', 'E', 'I']
path+[node]: ['A', 'C', 'D', 'H', 'I']
path+[node]: ['A', 'C', 'G', 'H', 'I']
path+[node]: ['A', 'C', 'D', 'E', 'H', 'I']
[['A', 'C', 'G', 'H', 'I'], 90]
```

目标设计：结果预示和功能介绍

- 功能介绍：
  - 如上一个幻灯片中展示可见，从上到下依次展示从起点A到终点I，会经过的所有可能的路径，可以看到有很多种情况。如倒数第四行到倒数第二行嘀倒数第一行的结果，虽然倒数第二行展示的所经路线更短，但考虑到堵车指数越低越好，因此倒数第一行的路线才是最佳路线。具体路线图在下一页。

功能介绍



导航

- 运用BFS算法，简单案例，在上下文上会有字母的不同，但不影响展示

导航

- 找到最佳路线
  - 假设A-C-G-H-I，为最佳路线，score=len(result[0])*20+(len(result[0]*50))/result[1]，该公式为计算最适宜路线的公式，score值越大，代表该路线越通畅。该公式的20-50的比例可以自行调整，具体看是距离优先还是堵车指数优先。
- 示例
  - 设定A为起点，H为终点，统计所有可能的路线和相对应的距离，以及与其对应的堵车指数。
  - 如下面数据举例所示，在综合考虑距离和堵车指数的情况下，最佳路线为A-D-E-G-H。
    - A-B-E-G-H 4 290
    - A-D-G-H 3 340
    - A-D-E-G-H 4 230
    - A-B-C-D-E-F-G-H 7 380

地图可视化

- 利用turtle模块
  - 用turtle模块为来绘制简略地图，
  - 距离计算是两个坐标之间的直线，会用到勾股定理，写好公式使用即可。该公式会在找出最佳路线的函数当中使用，就是findPath中
    - dist=(position[to][0]-position[current][0])**2+\
    - (position[to][1]-position[current][1])**2
    - dist=dist**0.5

地图可视化

- 利用turtle模块
  - 根据上面的数据，标号每个节点的位置，并连接节点，即可绘制一幅简单地图，规划好路线，最佳路线为A-D-E-G-H。

地图可视化

- 运用turtle来导航-预示（实验版-代码部分1）
  - 下面的代码部分并不是正式项目中写的代码，是为了学习和了解如何使用turtle来画图和导航。这部分代码是设置移动速度、对话框大小、线的粗细以及移动坐标时，会调用的方法等等。
    - # 引入turtle
    - import turtleturtle.speed(1)
    - # 该步骤是为了防止移动时出现走过的那条线，有需要这条线的时候，有不需要的时候，此时暂时不需要
    - turtle.up()
    - screen=turtle.Screen()
    - screen.setup(600,400)
    - # 向该坐标移动
    - turtle.goto(-200,0)
    - # 移动时划线
    - turtle.down()
    - turtle.color('aqua')
    - # 线的粗细
    - turtle.pensize(20)
    - turtle.goto(0,0)
    - turtle.goto(200,100)
    - # 若下一次需要划线的话，这步骤就不需要，不需要划线，则需调用up()
    - turtle.up()

地图可视化

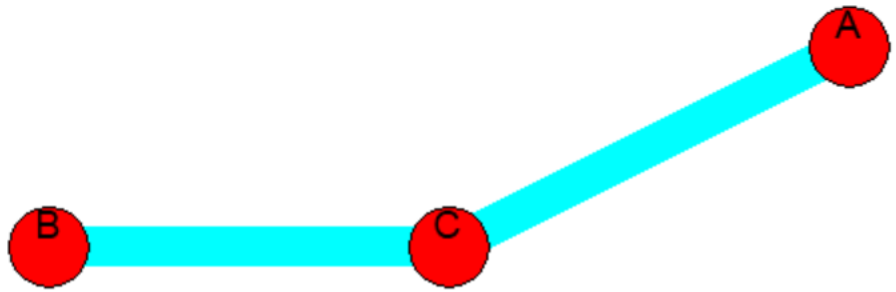
- 运用turtle来导航-预示（实验版-展示效果1）
  - 画线

地图可视化

- 运用turtle来导航-预示（实验版-代码部分2）
  - 模块样式设置
    - turtle.turtlesize(2)
    - turtle.shape('circle')
    - turtle.color('black','red')
    - # 在现在的位置留下痕迹
    - turtle.stamp()
    - turtle.write('A',align='center',font=('Arial',14))
    - turtle.goto(0,0)
    - turtle.stamp()
    - turtle.write('C',align='center',font=('Arial',14))
    - turtle.goto(-200,0)
    - turtle.stamp()
    - turtle.write('B',align='center',font=('Arial',14))
    - turtle.hideturtle()

地图可视化

- 运用turtle来导航-预示（实验版-展示效果2）
  - 模块样式设置

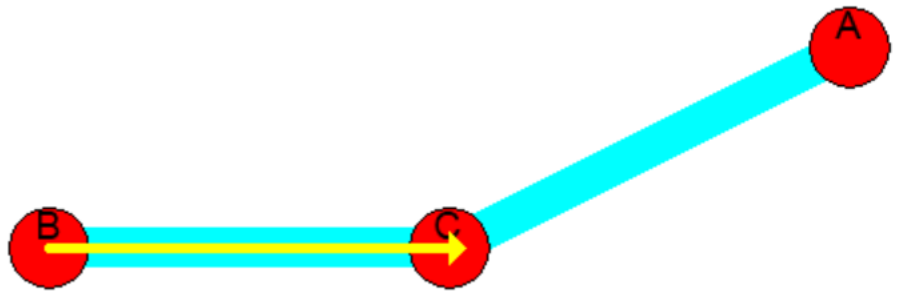


地图可视化

- 运用turtle来导航-预示（实验版-代码部分3）
  - 方向样式，最后的那行代码代表，启动Tkinter事件循环，保持绘图窗口持续显示，并等待用户交互。
    - turtle.turtlesize(0.8)
    - turtle.shape('arrow')
    - # 向该坐标移动，并画出向该坐标移动的指针
    - turtle.towards(0,0)
    - turtle.pensize(5)
    - turtle.color('yellow')
    - turtle.down()
    - turtle.showturtle()
    - turtle.goto(0,0)
    - # 启动 Tkinter 事件循环，保持绘图窗口持续显示，并等待用户交互
    - turtle.mainloop()

地图可视化

- 运用turtle来导航-预示（实验版-展示效果3）
    - 方向样式



地图抽象化

- 变量部分--将每个节点的坐标存储到字典当中
  - position=dic()
  - position[‘A’]=(-150,150)
  - position[‘B’]=(100,150)
  - position[‘C’]=(200,0)
  - position[‘D’]=(-125,0)
  - position[‘E’]=(25,0)
  - position[‘F’]=(150,0)
  - position[‘G’]=(25,-100)
  - position[‘H’]=(150,-100)

地图抽象化

- 变量部分--将节点之间的结构和堵车指数一起存储到名为graph的字典中，这部分代码的最后加上起终点的位置
  - graph=dic()
  - graph[‘A’]={ ‘B’ :90, ‘D’ :20}
  - graph[‘B’]={ ‘A’ :90, ‘E’ :50}
  - graph[‘C’]={ ‘D’ :60}
  - graph[‘D’]={ ‘A’ :20, ‘C’ :60, ‘E’ :30, ‘G’ :50}
  - graph[‘E’]={ ‘B’ :50, ‘D’ :30, ‘F’ :50, ‘G’ :15}
  - graph[‘F’]={ ‘E’ :50, ‘H’ :15}
  - graph[‘G’]={ ‘D’ :50, ‘E’ :15 , ‘H’ :25}
  - graph[‘H’]={ ‘F’ :50, ‘G’ :25}
  - start= ‘A’
  - end= ‘H’

地图抽象化

- 函数部分—def initialize，初始化数据的函数，设置turtle的样式。运行这部分代码后，会弹出长600像素，宽400像素的绘图窗口。
  - def initialize():
  -     turtle.speed(0)
  -     turtle.up()
  -     # 窗口样式
  -     screen = turtle.Screen()
  -     screen.setup(600, 400)
  -     # 修改窗口名
  -     screen.title('Path Finder')

地图抽象化

- 函数部分—def drawMap，利用turtle工具连接标出的每个点，遍历graph来实现

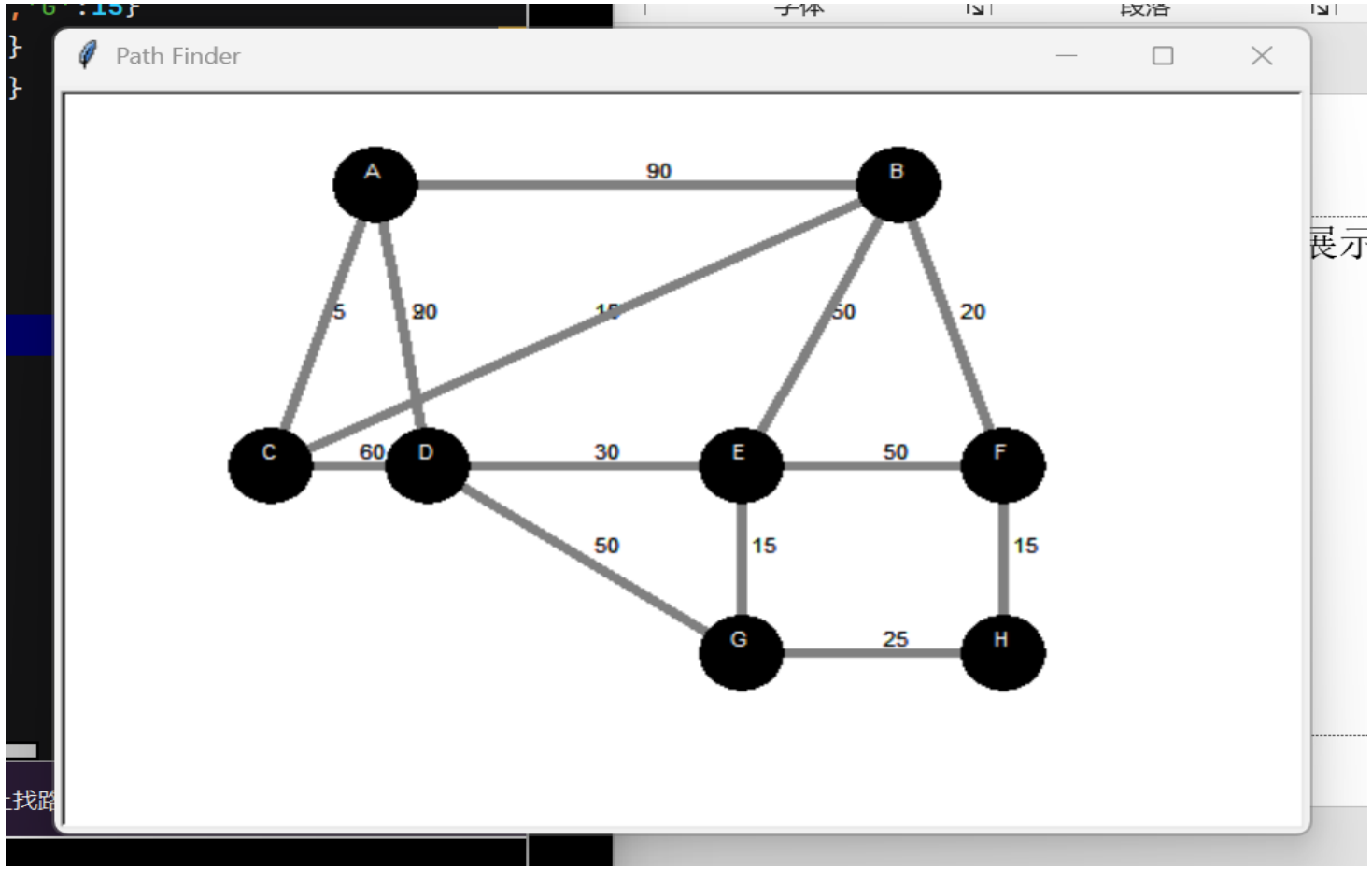
```
def drawMap():
    # 连接每个点
    turtle.pensize(5)
    turtle.color( ‘gray’)
    for start in graph:
        for finish in graph[start]:
            # 向start位置前进
            turtle.goto(position[start])
            # 放下笔，准备画图
            turtle.down()
            # 两个节点之间的中点坐标
            cx=position[start][0]+(position[finish][0]-position[start][0])/2
            cy = position[start][1] + (position[finish][1] - position[start][1]) / 2
            # 向中点坐标前进
            turtle.goto(cx,cy)
            turtle.color( ‘black’)
            # 坐标的名，就是A，B
            turtle.write( ‘  ’+str(graph[start][finish]))
            # 用灰色线连接每个节点
            turtle.color( ‘gray’)
            # 向终点前进
            turtle.goto(position[finish])
            # 抬起笔
            turtle.up()
```

地图抽象化

- 函数部分—def drawMap，通过遍历position，在绘图窗口标出每个节点的位置
  - def drawMap():
  -     # 画出每个点
  -     turtle.shape(‘circle’)
  -     turtle.turtlesize(2)
  -     # 遍历position字典
  -     for node in position:
  -         # print(‘position-node:’,node) # A B C D E F G H
  -         turtle.color('black')
  -         # 向每个点前进
  -         turtle.goto(position[node])
  -         # 在目前的位置留下痕迹
  -         turtle.stamp()
  -         # 白色字体
  -         turtle.color(‘white’)
  -         # 将node值写在圆形的中间
  -         turtle.write(node,align= ‘center’)
  -     turtle.hideturtle()

地图抽象化

- 函数部分—def drawMap，该部分效果图展示



地图抽象化

- 函数部分—def findPath，运用BFS算法找出最佳路线，这部分代码是通过BFS算法，找出最佳路线的，下一页的代码是同一个函数内的

```
def findPath(start,end): # 用上BFS
    # 队列初始化，用空列表来表示
    queue = []
    # 向队列中添加起点，移动的距离，堵车指数，目前经过的路线
    queue.append((start,0, 0,[start]))
    paths=[]
    while queue:
        current,distance,traffic,path=queue.pop(0)
        # print('current:',current)
        if current==end:
            # 将移动的距离，堵车指数以及所有可能的路径加入到paths中
            paths.append((distance,traffic,path))
            continue
        # 遍历目前所在点的堵车指数
        for to in graph[current]:
            # 若是没去过的地点
            if to not in path:
                # 此处运用了勾股定理，计算两点之间的直线距离
                dist=(position[to][0]-position[current][0])**2+ \
                     (position[to][1]-position[current][1])**2
                dist=dist**0.5
                # 则向queue中添加to，也就是要去的位置，从目前为止移动到目标位置移动的距离，堵车指数，以及从该节点能去的所有节点
                queue.append((to,distance+dist,traffic+graph[current][to],path+[to]))
                # print(to,distance+dist,traffic+graph[current][to],path+[to])
```

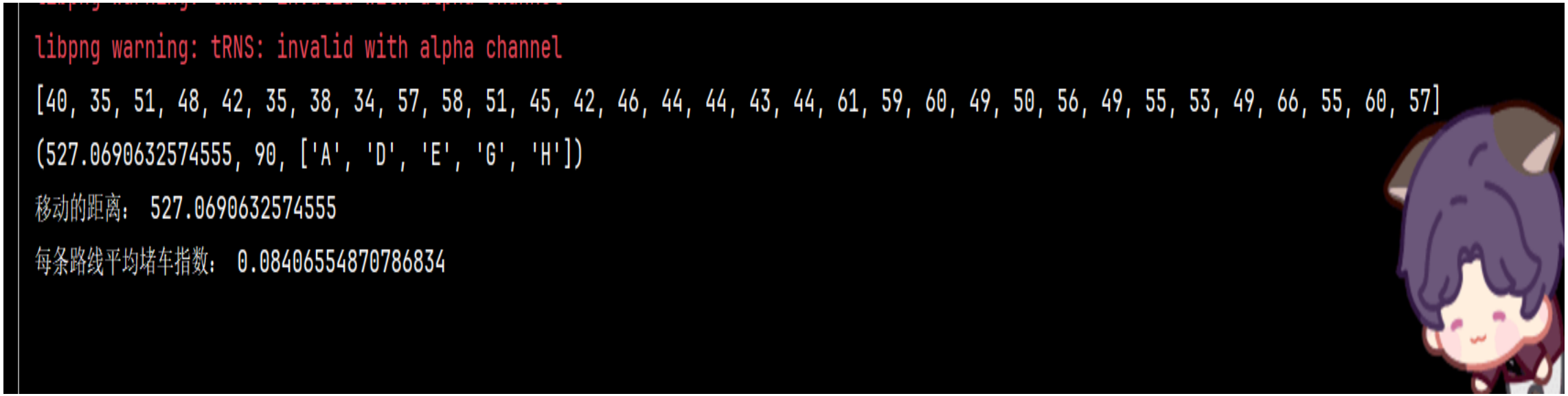
地图抽象化

- 函数部分—def findPath，运用BFS算法找出最佳路线，上一页代码是找出最佳路线的，这一页的代码是用勾股定理来计算起点到终点的直线距离，并用它来套用选出最佳路线的公式，距离除以起点到终点的直线距离，比重设置为15:85，这个比重可以随意设置，但15:85为最佳比例。

```
def findPath(start,end): # 用上BFS
    # 起点到终点的直线距离，用勾股定理
    directDistance=(position[end][0]-position[start][0])**2+ \(position[end][1]-position[start][1])**2
    directDistance = directDistance ** 0.5
    scores=[]
    mini=float('inf')
    # 遍历所有路径
    for result in paths:
        # result中包含距离，堵车指数和可能的所有节点
        # 距离除以起点到终点的直线距离
        d=result[0]/directDistance
        # 寻找最佳路径公式，需要注意比重，20:50
        # score=d*20+(result[1]/d)*50
        # 这样的公式是15:85为最佳路径比例公式
        score=d*15+(result[1]/result[0])*85
        score=int(score)
        scores.append(score)
        # 若设置的mini大于score，则将score的值赋值给mini
        if mini>score:
            mini=score
    return paths[scores.index(mini)][2]
```

地图抽象化

- 函数部分—def findPath，运行findPath函数结果，在会话窗口的效果与drawMap函数的结果并无不同，因为该函数是为了计算和筛选出最佳路线的的数值的，下图中展示的就是若干路线中最适合出行的路线，及其移动距离和堵车指数值。



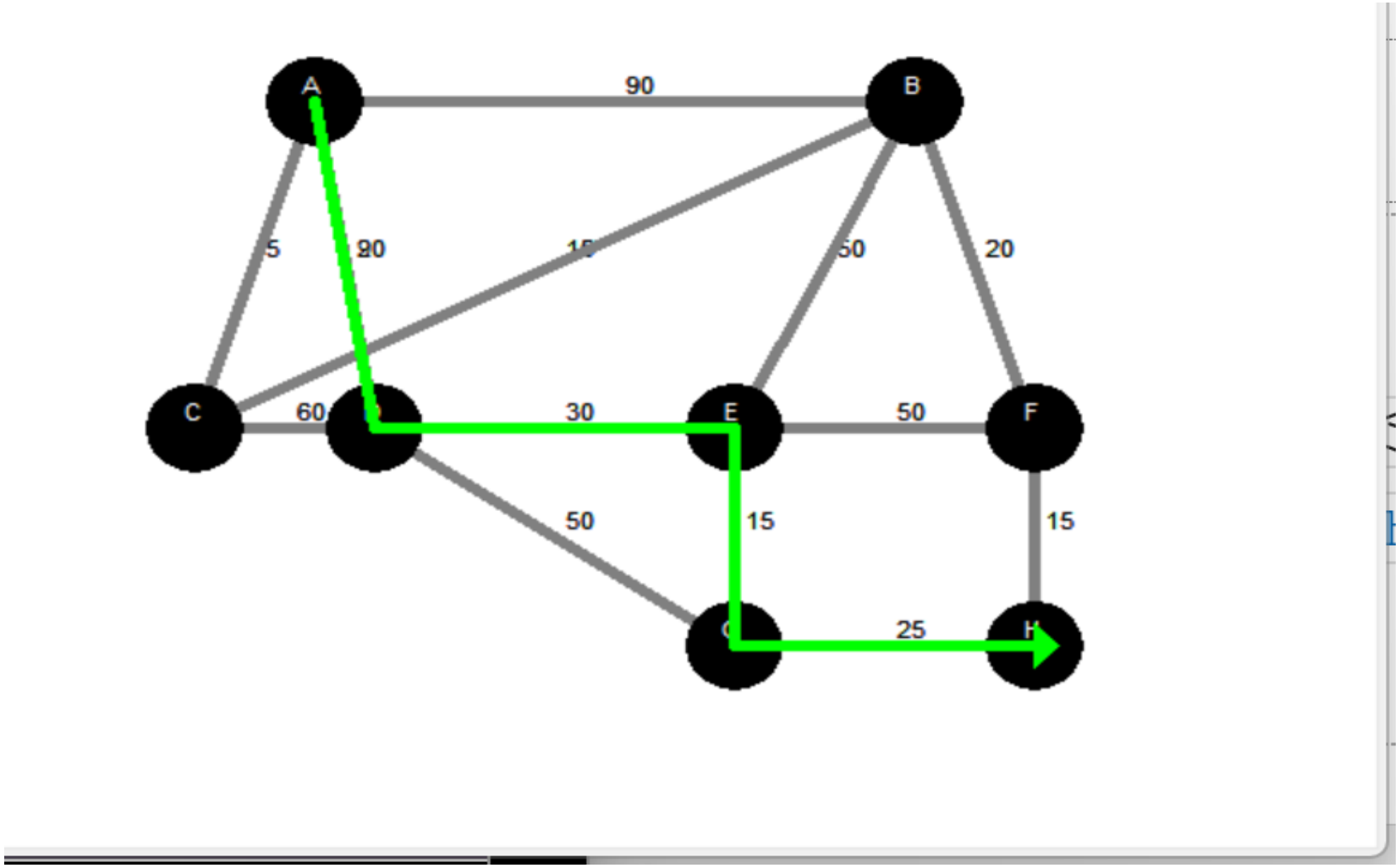
地图抽象化

- 函数部分—def drawPath，用lime颜色的线标出最佳路线。
  - turtle.pensize(5)
  - turtle.color('lime')
  - # 遍历所有路径的长度-1
  - for index in range(len(path)-1):
  - # 让海龟移动到 path 路径中第 index 个点所对应的坐标位置。
  - turtle.goto(position[path[index]])
  - turtle.speed(1)
  - turtle.down()
  - # 让海龟移动到 path 列表中下一个点（index + 1）
  - turtle.goto(position[path[index+1]])
  - turtle.up()
  - turtle.shape('arrow')
  - turtle.showturtle()
  - turtle.turtlesize(1)

地图抽象化

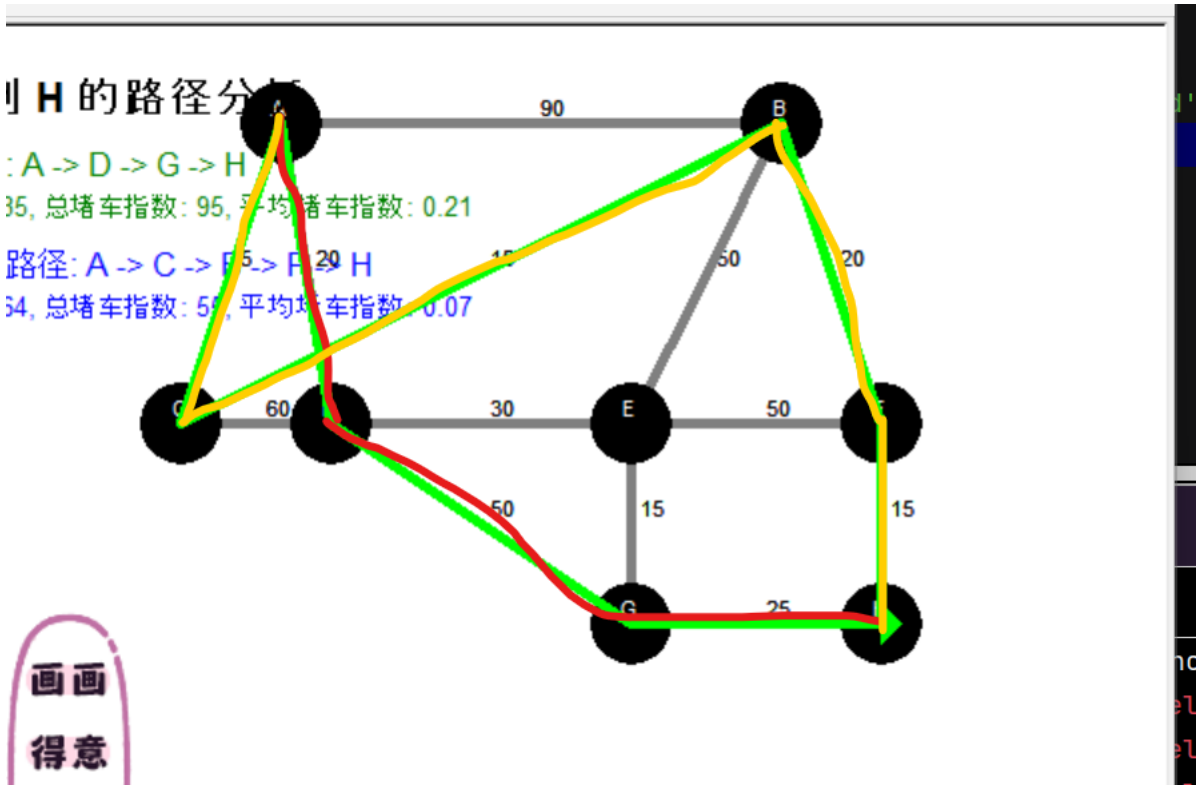
- 函数运行部分
  - initialize()
  - drawMap()
  - path=findPath(start,end)
  - drawPath(path)
  - turtle.mainloop()

地图抽象化最终效果



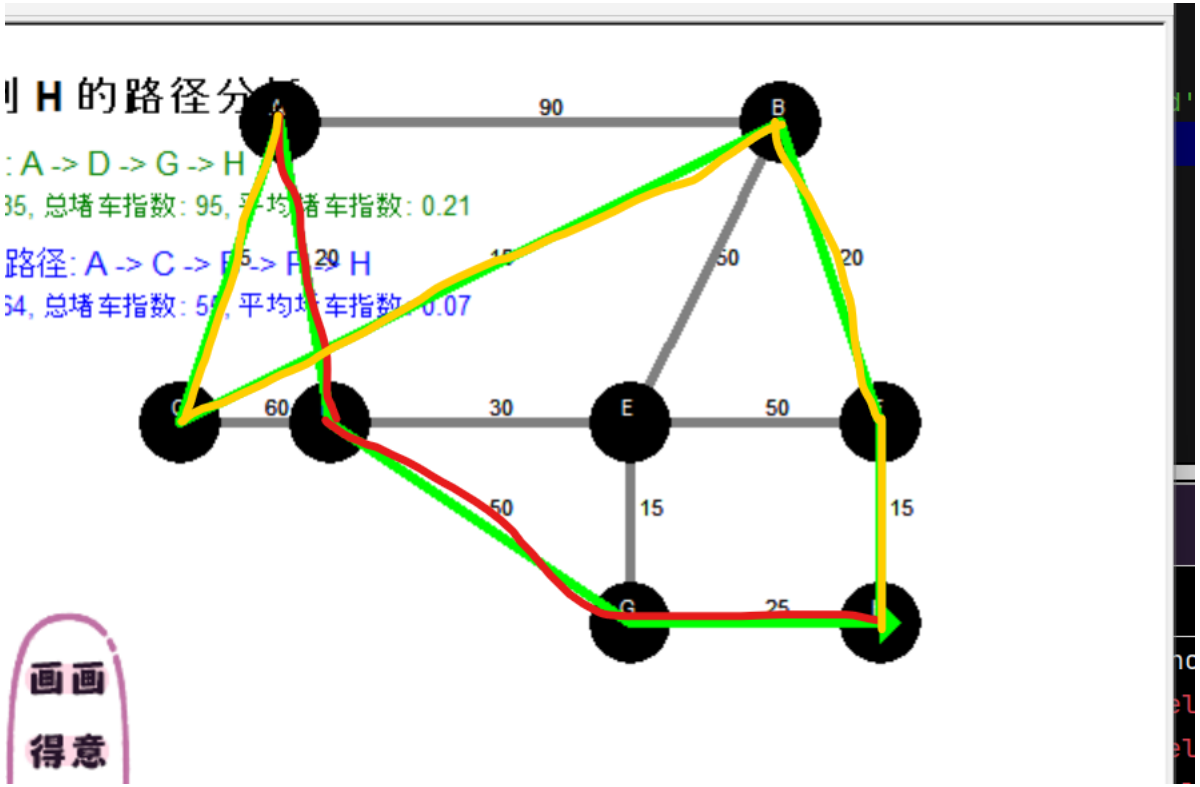
最短距离——与最佳路线比较

```
(508.11388300841895, 125, ['A', 'B', 'F', 'H'])
(457.346627030655, 95, ['A', 'D', 'G', 'H'])
(642.7050983124842, 205, ['A', 'B', 'E', 'F', 'H'])
(642.7050983124842, 180, ['A', 'B', 'E', 'G', 'H'])
(538.3914467816185, 140, ['A', 'C', 'D', 'G', 'H'])
(751.6379626418064, 55, ['A', 'C', 'B', 'F', 'H'])
(527.0690632574555, 115, ['A', 'D', 'E', 'F', 'H'])
(527.0690632574555, 90, ['A', 'D', 'E', 'G', 'H'])
(872.9826620856836, 245, ['A', 'B', 'E', 'D', 'G', 'H'])
(965.6877603981679, 240, ['A', 'B', 'C', 'D', 'G', 'H'])
(758.113883008419, 200, ['A', 'B', 'F', 'E', 'G', 'H'])
(608.113883008419, 160, ['A', 'C', 'D', 'E', 'F', 'H'])
(608.113883008419, 135, ['A', 'C', 'D', 'E', 'G', 'H'])
(886.2291779458717, 135, ['A', 'C', 'B', 'E', 'F', 'H'])
(886.2291779458717, 110, ['A', 'C', 'B', 'E', 'G', 'H'])
(820.593142890843, 130, ['A', 'D', 'C', 'B', 'F', 'H'])
(727.8880445783586, 135, ['A', 'D', 'E', 'B', 'F', 'H'])
(657.346627030655, 150, ['A', 'D', 'G', 'E', 'F', 'H'])
(1035.4101966249684, 260, ['A', 'B', 'C', 'D', 'E', 'F', 'H'])
(1035.4101966249684, 235, ['A', 'B', 'C', 'D', 'E', 'G', 'H'])
(988.3914467816185, 265, ['A', 'B', 'F', 'E', 'D', 'G', 'H'])
(808.9328643293221, 180, ['A', 'C', 'D', 'E', 'B', 'F', 'H'])
```



最不堵车路线—与最佳路线比较

```
(642.7050983124842, 180, ['A', 'B', 'E', 'G', 'H'])
(538.3914467816185, 140, ['A', 'C', 'D', 'G', 'H'])
(751.6379626418064, 55, ['A', 'C', 'B', 'F', 'H'])
(527.0690632574555, 115, ['A', 'D', 'E', 'F', 'H'])
(527.0690632574555, 90, ['A', 'D', 'E', 'G', 'H'])
(872.9826620856836, 245, ['A', 'B', 'E', 'D', 'G', 'H'])
(965.6877603981679, 240, ['A', 'B', 'C', 'D', 'G', 'H'])
(758.113883008419, 200, ['A', 'B', 'F', 'E', 'G', 'H'])
(608.113883008419, 160, ['A', 'C', 'D', 'E', 'F', 'H'])
(608.113883008419, 135, ['A', 'C', 'D', 'E', 'G', 'H'])
(886.2291779458717, 135, ['A', 'C', 'B', 'E', 'F', 'H'])
(886.2291779458717, 110, ['A', 'C', 'B', 'E', 'G', 'H'])
(820.593142890843, 130, ['A', 'D', 'C', 'B', 'F', 'H'])
(727.8880445783586, 135, ['A', 'D', 'E', 'B', 'F', 'H'])
(657.346627030655, 150, ['A', 'D', 'G', 'E', 'F', 'H'])
(1035.4101966249684, 260, ['A', 'B', 'C', 'D', 'E', 'F', 'H'])
(1035.4101966249684, 235, ['A', 'B', 'C', 'D', 'E', 'G', 'H'])
```



总结

- 对于turtle和BFS的了解和运用不怎么到位
- 节点不多时，用上面结合了BFS算法的方法，但节点比现在更多，要搜索的范围更广时，BFS就不再是最优选的方法，除了BFS算法还有Dijkstra算法。
- 但范围更加广泛时，会连Dijkstra算法都会不适用，此时可以使用A*算法，A*算法的特点就是会无视掉不必要的或更远的路线，会直接获取最短的路线。

难点与重点

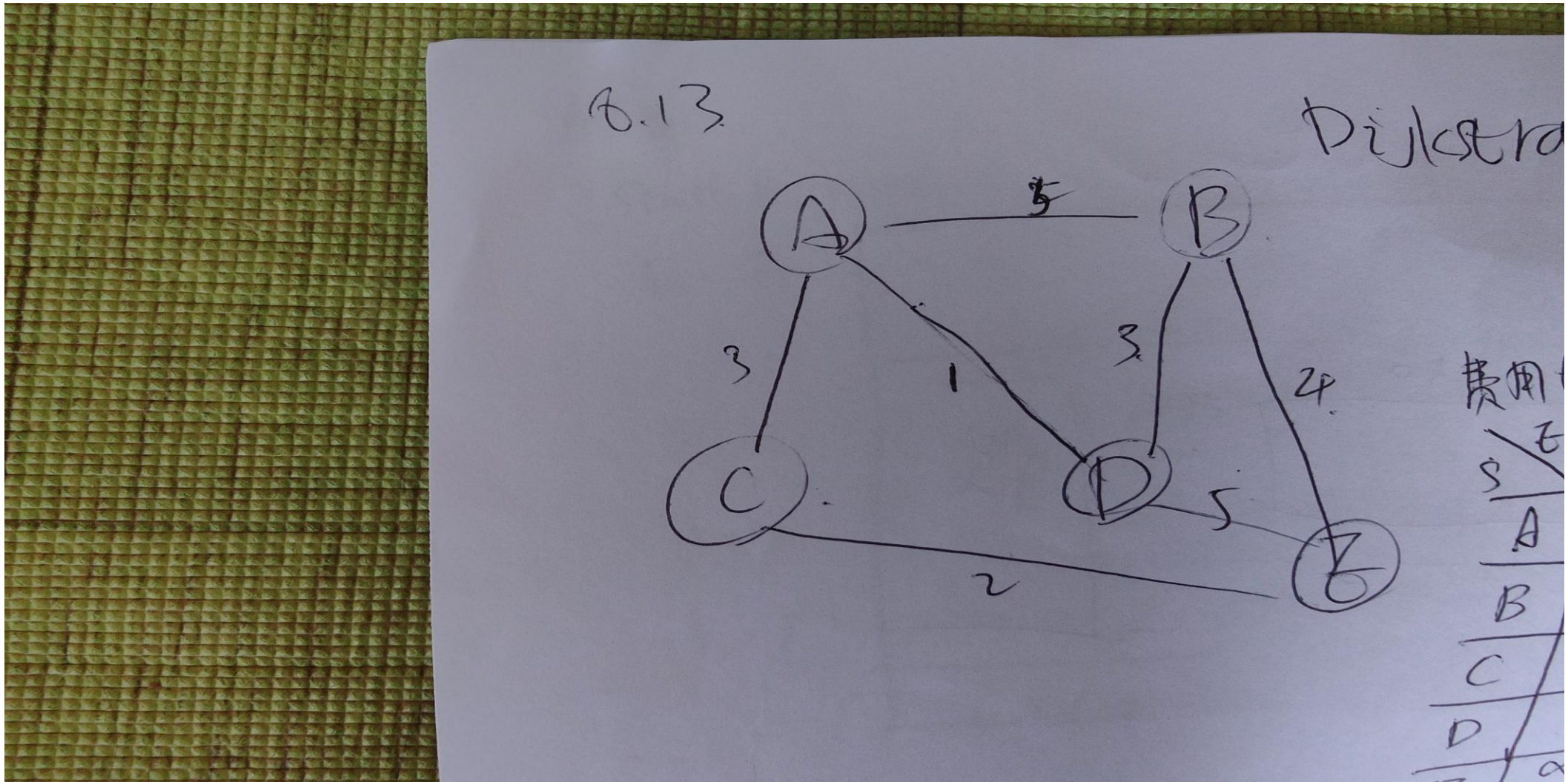
- BFS算法结合turtle绘制节点和连接每个节点时，没能完全融会贯通。
- 不同于用BFS算法只在控制台输出最佳路线的节点，以及总距离和堵车指数，想要将这些节点绘制到绘图窗口，并且还要计算中点值来辅助连接每个节点。
- 能否更加熟练流畅地使用turtle库和库中的方法。

感受

- 需要能更加熟练流畅地使用turtle库和库中的方法。
- 需要多加练习运用到BFS算法的多种案例。

Dijkstra算法

- 无方向、无指定起终点的简略地图



A*算法

- 运用A*算法的话，设置好起终点后，无论地图的范围有多大、多广，算法会默认自动在起终点所在的最小范围内搜查最短路线，屏蔽掉不必要计算的太远的路线。这区别于BFS算法，BFS算法是无论距离远近，只要能符合，就会列出所有可行的路线和方法。
- 下面两幅图是手动绘制的简略图，老师画的和我画的。假设地图的范围更广，且起点和终点之间还有一个障碍物，但用A*算法的话，就屏蔽掉了更绕的路线，直接锁定起点到终点之间最小的范围，并在这范围当中，根据要求筛选出最短或最佳路线。在图片中找的是最短路线。

A*算法

- 直接上图比较方便，左边是老师演示的，右边是我画的